Designing a simulator for POOSL

I. Mankoe

Author : I. Mankoe
Coaches : ing. P.H.A. van der Putten
          ir. J.P.M. Voeten
Supervisor : Prof.ir. M.P.J. Stevens
Period : January 1996 - September 1996

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of student projects and graduation reports.
Abstract

Within the Information and Communication Systems Group at the Eindhoven University of Technology, active research is done into design methods and tools that support the development of complex systems. This research resulted in the object-oriented design methodology SHE (Software/Hardware Engineering). As a part of SHE the formal specification language POOSL (Parallel Object-Oriented Specification Language) is designed. Starting from informal graphical modelling the SHE method produces rigorous specifications described in the POOSL language. Once a POOSL specification of a system is made, it can be used to automate subsequent design steps. To do so, one has to be sure that the specification is correct. For this reason the need for a tool arises which can be used to check the correctness of a specification: a POOSL simulator.

In this thesis the first steps towards the design and implementation of a POOSL simulator are described. The mapping of POOSL onto the programming language Smalltalk and the conversion of the POOSL syntax into Smalltalk equivalents are the main subjects of this thesis.

First an inventory is made of the differences between the specification language POOSL and the programming language Smalltalk. This is followed by the mapping problems that result from these differences. Solutions for these problems are given, resulting in conceptual solutions for the POOSL simulator.

Special attention is paid to the design and implementation of a mechanism in Smalltalk, which is used to simulate the communication between POOSL process objects. The conversion of the POOSL syntax into its Smalltalk equivalent is described in detail. Conversion rules for almost all process statements are given.

Finally, the implementation and conversion rules are used to simulate a test design in order to demonstrate the resulting implementation.
Table of contents

Chapter 1 Introduction ........................................... 5
  1.1 Context of research ........................................... 5
  1.2 Problem definition ........................................... 5
  1.3 Deliverables .................................................. 6
  1.4 Organization of the thesis ................................... 6

Chapter 2 On the design of a simulator ............................ 7
  2.1 The basics of the POOSL language ............................ 7
  2.2 The Smalltalk language ...................................... 7
  2.3 The mapping problem ......................................... 9

Chapter 3 Conceptual solution ..................................... 12
  3.1 Mapping of POOSL onto Smalltalk ............................ 12
  3.2 Conceptual communication design ............................ 13
    3.2.1 Process requests ....................................... 14
    3.2.2 Modelling the communication channels .................. 14
    3.2.3 Requests invalidation .................................. 14
  3.3 The scheduler ................................................ 15
    3.3.1 Scheduled requests ..................................... 15
    3.3.2 Requests to schedule ................................... 15
    3.3.3 Simulation time ......................................... 16
  3.4 Causally related requests ................................... 16
  3.5 Guarded requests ............................................. 18
  3.6 Statement request ........................................... 18
  3.7 Modelling of process objects ................................ 19
    3.7.1 Process mailBox ......................................... 19
    3.7.2 Simulator execution states .............................. 19

Chapter 4 Implementation of the simulator ....................... 21
  4.1 The POOSL class .............................................. 21
    4.1.1 Class PChannel .......................................... 23
    4.1.2 Class PData ............................................. 23
    4.1.3 Class PProcess .......................................... 23
    4.1.4 Class PRequest .......................................... 23
      4.1.4.1 Class PComRequest .................................. 24
      4.1.4.2 Class PWaitRequest .................................. 24
      4.1.4.3 Class PSendRequest .................................. 25
      4.1.4.4 Class PReceiveRequest .............................. 25
      4.1.4.5 Class PScheduledComReq ............................. 26
      4.1.4.6 Class PStatementRequest ............................ 26
    4.1.5 Class PScheduler ........................................ 27
4.2 Handling communication requests ........................................... 27
4.3 Simulator input ..................................................................... 30
4.4 Simulator output .................................................................. 30

Chapter 5 Smalltalk equivalents for POOSL syntax ......................... 32
5.1 Process statements .................................................................. 32
5.1.1 Message-send .................................................................... 33
5.1.2 Message-receive .............................................................. 35
5.1.3 method-call ...................................................................... 37
5.1.4 sequential composition ..................................................... 38
5.1.5 select statement ............................................................... 39
5.1.6 if-statement ..................................................................... 41
5.1.7 guarded command ............................................................ 42
5.1.8 while-statement ............................................................... 44
5.2 Process class .......................................................................... 45
5.3 Data objects .......................................................................... 47

Chapter 6 Test case ..................................................................... 49
6.1 Specification .......................................................................... 49
6.2 Simulation results ................................................................. 49

Chapter 7 Future research ............................................................ 51

Chapter 8 Conclusions .................................................................. 52

References ................................................................................. 53

Appendix 1 The POOSL class hierarchy ........................................ 54

Appendix 2 Specification of test design ....................................... 65

Appendix 3 Simulation results ...................................................... 71
Chapter 1

Introduction

1.1 Context of research

The design of current information-technology products becomes more and more difficult because of their growing complexity. Products often contain information processing systems that consist of a combination of software and hardware components and that have properties that characterize them as parallel, hierarchical, distributed, real-time and/or embedded systems. To support the development of these complex systems, good analysis, specification and design methods and tools are required.

The development of these methods and tools is the subject of active research within the Information and Communication Systems Group at the Eindhoven University of Technology. This research has resulted in the SHE (Software/Hardware Engineering) method [PVS95]. SHE is an object-oriented method for the co-development of complex hardware/software systems, covering (co-)analysis, (co-)specification and (co-)design. The method incorporates the formal description language POOSL (Parallel Object-Oriented Specification Language) [Voe95a, Voe95b].

1.2 Problem definition

SHE starts from informal graphical analysis and produces formal specification described in the POOSL language. The formalization in POOSL will be used in future to support the use of automated design tools.

An important tool that has to be developed is a simulator tool. Such a tool is needed to detect failures in an early stage and to verify the behaviour of POOSL specifications. Using this simulator a designer must be able to verify the design by analysing the behaviour of the specification. The designer must be able to make changes during the simulation, which makes it easier to investigate different possibilities. The simulator has to be able to simulate all valid POOSL specifications. Further, its appearance must be like the IDaSS simulator [Ver90]. This means that a user must be able to actually draw a behaviour model of the design, must be able to retrieve all simulation information and walk through the simulation step by step. Other features are for instance the storing of designs in a file, the reading of designs from a file and automatic documentation generation.
Introduction

1.3 Deliverables

Implementing a good and extensive simulator, especially one of the kind as described in the previous section, takes quite some time. It can easily take a few years before a satisfying version is implemented. It is of importance to divide the implementation in phases. The work discussed in this thesis is a start for the implementation of the simulator for POOSL. In this first phase of the implementation there are two important aspects:

- Mapping of POOSL specifications onto Smalltalk equivalents. The issues here are how to represent and implement the different parts of POOSL, such as process objects, data objects, channels, and communications in the Smalltalk environment.

- Defining conversion rules for the conversion of POOSL statements into Smalltalk equivalents. These rules are necessary because some of the statements used in POOSL are unusual or totally unknown in Smalltalk.

1.4 Organization of the thesis

This thesis is divided into 8 chapters. In chapter 2 the POOSL specification language and the Smalltalk programming language are discussed briefly. The problems of mapping POOSL onto Smalltalk are also subject of this chapter.

In chapter 3 the conceptual solutions of the mapping and the problems encountered are given.

In Chapter 4 the implementation of the simulator is described. The class hierarchy of the POOSL simulator is given and the implementation of the communication is described in detail.

The conversion rules of the POOSL syntax into Smalltalk equivalents are given in chapter 5. The conversion of each POOSL process statement is illustrated with an example.

In chapter 6 the simulator designed and implemented in this phase is tested. The test design that was used to test the non-deterministic selecting behaviour of the simulator. The simulation results are presented.

In chapter 7 the continuation of the design and implementation of the POOSL simulator is discussed.

Conclusions regarding the design and implementation of this first phase towards a full simulator for POOSL are given in chapter 8.
Chapter 2

On the design of a simulator

2.1 The basics of the POOSL language

POOSL (Parallel Object-Oriented Specification Language) [Voe95a, Voe95b] is a specification and design language which is developed as a part of an object-oriented methodology for the specification and design of data processing systems. The language is based on the object-oriented paradigm to support flexible and reusable design, as well as on the basic concepts of CCS [Mil80, Mil89] to enable formal verification, simulation, and transformation of specifications.

A specification in POOSL consists of a fixed set of concurrent process objects and process-object clusters, which are statically interconnected with channels through which they can communicate by exchanging messages. Message exchange is based upon the synchronous pair-wise message-passing mechanism (rendez-vous) of CCS.

When a process wants to communicate, it explicitly states to which channel this message has to be sent and when and from which channel it wants to receive a message. A communication can only take place if two processes are willing to communicate with each other. A process indicates that it is willing to communicate by executing a message-send or message-receive statement. A process is also able to select a communication collaborator out of a number of alternatives.

Besides processes and clusters, POOSL also supports data objects. A data object consists of some private data and has the ability to act on this data. Data is stored in instance variables, which contain (references to) others objects or to objects which own the variables. Variables of an object cannot be accessed directly by any other object. They can only be read and changed by the object itself.

2.2 The Smalltalk language

Smalltalk [Dig91] is a sequential object-oriented programming language. Every entity within the Smalltalk environment is an object. Smalltalk offers a standard class hierarchy, which can be used by the user. The object-oriented nature of Smalltalk requires a class hierarchy for each application implemented in Smalltalk.
On the design of a simulator

A class defines the behaviour of similar objects by specifying the variables they contain and the methods that are available for responding to messages sent to them. To define a class specification in Smalltalk the next protocol is used:

```
SuperClassName subclass: #className
instanceVariablesNames:instanceVariableNameString
classVariableNames:classVariableNameString
class methods
instance methods
```

`SuperClassName` is the name of the superclass which is subclassed by the class defined. `className` is the name of the class defined. The `instanceVariableNameString` is a string containing the instance variables of the class. They are the internal variables of an object and contain references to other objects. They are private to the object in which they are defined and can not be accessed directly by other objects. If an object wants to change a variable of another object, it does this by sending a message to the owning object. The class variables must be defined in the string `classVariableString`. These are the so called shared data of the class and can be accessed by both instance methods and class methods.

Next, the `class methods` must be given. These methods can be sent only to the class name. Class methods are typically used to initialize class variables, to create instances of the class, or to provide general inquiries without instance creation.

At the end the `instance methods` of the class are defined. The `instance methods` of the owning object determine which operations can be performed on the instance variables. A method is a piece of Smalltalk code specifying the object's behaviour.

Smalltalk objects communicate with each other by sending messages to each other. The messages they send consist of a selector and zero or more arguments. The selector represents the name of the corresponding method and the arguments are the parameters passed to it. Suppose an object sends the following message:

```
multiply:aNumber with:anotherNumber
```

In this case `multiply:with:` is the selector, `aNumber` and `anotherNumber` are the arguments passed along with the message. The selector must correspond to a method of the object to which this message is sent, otherwise Smalltalk will respond with an error message. Selectors can be compared to function names of procedural programming languages such as `PASCAL` or C. Messages can be compared to the calls of these functions.

With the class hierarchy the inheritance of methods and variables between the classes and the subclasses is stated. Inheritance is a powerful tool in object-oriented programming; a class inherits all the methods and variables of its predecessor class in the class hierarchy. An example of a class hierarchy is given in figure 2.1.
The class Animal is called the superclass of the given animal class hierarchy. It is used to describe the general behaviour of all animals in this class hierarchy. The classes Bird and Mammal are subclasses of Animal that inherit the class behaviour described in Animal. The class Bird on its turn has the subclasses Parrot and Penguin, specifying the specific behaviours of these groups of animals. They inherit the behaviour described in class Bird and as well as the behaviour described in class Animal, the superclasses of Parrot and Penguin.

It is also possible to start a process in Smalltalk. A Smalltalk process is a sequence of computation carried out by sending messages to other objects and waiting for the results. A process has a name and a priority. A new process is created and becomes ready to run as a result of sending the fork or fork:at: message to a block. This means that the process is waiting until the processor is assigned to it. A ready process will become active if it is the longest waiting process and if there are no other waiting processes with a higher priority. The activation of a ready process will take place when the active process becomes blocked or dead, or if the ready process has a higher priority than the active process, or if the ready process has the same priority as the active process and the expression Processor yield is executed.

Processor is an instance of class ProcessScheduler and is maintained as a global variable within Smalltalk. The class ProcessScheduler is a member of the standard class hierarchy of Smalltalk. If the message yield is sent to it, it gives other processes at the priority of the currently running process a chance to run.

Smalltalk/V has many built in objects, many of which are contained in global variables. Unlike temporary variables, Smalltalk does not automatically dispose global variables when you are finished using them, and their use is not confined to a single set of expressions. For example, Smalltalk/V provides (among others) the following global variables: Display, Transcript, Disk and Smalltalk.

2.3 The mapping problem
If we want to design a simulator for POOSL we have to design a mapping of POOSL specifications onto Smalltalk equivalents. This mapping problem is visualized in figure 2.2.

![Figure 2.2 The mapping problem.](image)

The mapping lays the foundation for the simulator. It is of importance that the mapping covers all features of POOSL. The better the mapping, the better the implementation of the simulator.

If we want to map POOSL specifications onto Smalltalk equivalents, we will encounter a number of problems. These problems are caused by the essential differences between POOSL and Smalltalk. In the following, the mapping problems are discussed one by one.

The main problem of the mapping is the communication principle of POOSL. Communication in POOSL is *synchronous, pair-wise*, and uses communication channels. *Synchronous* means that two process objects only communicate if one is executing a *message-send* statement and the other is simultaneously executing a corresponding *message-receive* statement. *Pair-wise* means that different process objects can be connected to one and the same channel, but a process connected to that channel can at any time communicate with only one process connected to that channel.

In Smalltalk no channels are used and the receiver of the message is always ready to communicate. Communication between Smalltalk objects is synchronous. The object sending a message suspends its execution until it receives a reply from the receiving object. In Smalltalk, the receiving object always replies with a single object as a result to the sending object. Because there are no channels available in Smalltalk a concept is needed to model the channels of a POOSL specification onto a Smalltalk equivalent.

In POOSL, different process objects are allowed to communicate at the same time. Further, communications in POOSL are time related; a channel requires time to transmit messages.
from one process to another. In Smalltalk, however, communications between objects are executed sequentially, and the time needed for the communication has no relation with the transmission time of the channels in POOSL.

If we want to model the transmission time of a channel in Smalltalk, a mechanism must be designed to indicate that a channel is in use. If the transmission time is elapsed, it must release the channel for other communications using the channel.

This includes that a mechanism is needed in the Smalltalk equivalent to control the simulation time. Further, a prescription must be given of how to map the parallel communication in Smalltalk.

A POOSL process can select one communication collaborator out of a number of alternatives. The process leaves all alternatives open, until one collaborator is ready to communicate. Once a collaborator is prepared to communicate, the other alternatives become invalid. This means that the Smalltalk equivalent must be provided with the opportunity to monitor all message-send and message-receive statements of one and the same process. Further, it must be able to withdraw messages which are no longer valid.
Chapter 3

Conceptual solution

In the previous chapter the problems for the mapping of POOSL onto Smalltalk were discussed. In this chapter conceptual solutions for those problems are given.

3.1 Mapping of POOSL onto Smalltalk

To simulate a POOSL description, a mapping of the description onto a Smalltalk equivalent is needed. With this mapping the relationship between POOSL objects and the equivalent Smalltalk objects is stated.

The general idea of the mapping is shown in figure 3.1. For each process object in the POOSL description an equivalent Smalltalk object is created and a Smalltalk process is started. For each data object in POOSL a corresponding Smalltalk object is created. The communication channels defined in the POOSL description are also represented by Smalltalk objects. The message-send and message-receive statements in POOSL are represented by request objects in the Smalltalk equivalent.

To simulate the Smalltalk equivalent, an additional object is needed to complete the Smalltalk equivalent: a scheduler. The scheduler is used to model and to control the time and the communication in the Smalltalk environment. The different parts of this mapping are subjects of the following sections.

Figure 3.1 Mapping of POOSL onto Smalltalk.
3.2 Conceptual communication design

The basic idea to handle communication in POOSL is that whenever a process wants to communicate, it executes a *message-send* or a *message-receive* statement. It explicitly states which channel it wants to use to send or receive a message. If another process is willing to communicate, the communication is executed.

In the Smalltalk equivalent the execution of the *message-send* and *message-receive* statements are modelled by request objects. Execution of one of these statements in the POOSL environment is modelled in the Smalltalk equivalent by creating a request object and sending it to the channel used. The channel collects all the requests received, and will generate a request to communicate, if two matching requests are received. Two requests are matching if one request is a request to send, and the other request is a request to receive. Further, the names of both requests must be the same message name. The created communication request is sent to the scheduler. The scheduler takes care of the scheduling and the execution of the communication request received. This is illustrated in the figure 3.2. In the following subsections the different parts of the Smalltalk model are discussed.

![Diagram](image)

*Figure 3.2 Smalltalk model for communication between processes.*
3.2.1 Process requests

The different statements used for communications and delays in POOSL are represented by different request objects in Smalltalk. The execution of these statements are modelled by creating an instance of the appropriate request and sending it to a channel or to the scheduler.

Requests representing *message-send* or *message-receive* statements are sent to the channel used for the communication. They cannot be sent directly to the scheduler, because the collaborator is not known at the moment the process object performs the request. The channel is responsible for pairing the requests of two collaborating processes. This is discussed in the following section.

Requests not representing *message-send* or *message-receive* statements can be sent directly to the scheduler. This can be done because no collaborator is required for the execution of these requests.

3.2.2 Modelling the communication channels

If a POOSL process object wants to communicate, it uses a channel to send or to receive messages. A communication between two processes can only take place if both are willing to communicate. Further, one of them must act as a sender and the other as receiver and they must use the same channel. Channels are unknown in Smalltalk, thus we have to give a design to model a channel used in POOSL into a Smalltalk equivalent. The channels are modelled as Smalltalk objects.

Process objects can send their requests to the channel. The channel will collect these requests and for each newly received request the channel will search for requests that match with it. If a match is found, the channel creates an object representing a communication between two processes and sends it to the scheduler. The scheduler takes control of the further handling of the communication.

If a channel is used for communication no other communication can be executed using the same channel at that time. To prevent that two communications are executed at the same time using the same channel, the channel is provided with a variable indicating its status. A channel can either be blocked, meaning that a communication is going on, or unblocked. A communication can be started only when the channel used for communication is free (unblocked).

3.2.3 Requests invalidation

The requests performed by the process objects are no longer useful for the simulation if they are executed. These requests are called invalid requests. Objects holding references to invalid requests can remove them from their administration. To do so, an object must know when a request can be removed from its administration. For this purpose the classes of the requests are provided with an instance variable indicating whether a request is still
valid or not. Initially the value of this variable is valid and it is set to invalid if the request itself is executed or a causally related request is executed. Causally related requests are discussed later on in this chapter. The objects holding one or more requests can inspect their validity.

3.3 The scheduler

The scheduler is a Smalltalk object used to model and to control the time in the Smalltalk equivalent. It also controls execution of the communications and the simulation progress. In POOSL time is hidden in the description. Its presence is revealed because processes are able to execute statements which are time consuming or time related. For example, a process can execute a statement to wait for some time before it resumes with its execution. Also the transmission time of channels indicates the presence of time hidden in a POOSL description.

To carry out its tasks, the scheduler is provided with a variable representing the simulation time and two sets; one is used to store the requests which are not scheduled yet and one to store the requests which are scheduled to be executed. We will call these sets requestsToSchedule and scheduledRequests respectively. In the following subsections these elements of the scheduler are discussed.

3.3.1 Scheduled requests

To store requests which are scheduled to be executed, the scheduler uses the set scheduledRequests. This set is organised in a special way. It is a collection with sets as elements. The elements of these set are requests scheduled for execution. The sets in scheduledRequests are ordered by a key. The key represents the time at which the requests of a set must be executed. Each request added to scheduledRequests is added to the set whose key is equal to the time at which the request must be executed.

This organization of scheduledRequests makes it easy for the scheduler to select the requests scheduled for execution. If the scheduler asks for the requests that have to be executed next, the set with the smallest key is passed to it.

3.3.2 Requests to schedule

The set requestsToSchedule is used to store incoming requests and requests which are not scheduled yet. Process and channel objects are free to add their requests to this set. When the scheduler is ready to service the requests in this set, the invalid requests in this set are removed first. Next, the scheduler selects a request from requestsToSchedule and sends the message execute to it.

If the selected request is finished with the execution of the corresponding method, the scheduler selects a new request from requestsToSchedule and the message execute is sent to it.
This will be repeated until no valid requests, which can be executed at the present simulation time, are left in requestsToSchedule.

3.3.3 Simulation time

The simulation time gives an indication of the simulation duration based on the time needed to perform communications and delays. The execution of statements that are not related to communications or delays, are not supposed to take time. Therefore the execution of these statements has no effect on the simulation time.

If a new simulation is started and initialized, the simulation time is set to zero. The scheduler starts to handle the requests in scheduledRequests and requestsToSchedule. If there are no more requests left in these sets which can be handled by the scheduler at the present simulation time, the simulation time is adjusted to the key value of the first set in scheduledRequests. The procedure of handling requests and adjusting the simulation time is repeated until the moment is reached at which both requestsToSchedule and scheduledRequests are empty sets. This indicates that no new requests will be performed so that the simulation is ended.

3.4 Causally related requests

A POOSL process object is able to do different requests at the same time if it executes a select-statement. Only one of these requests can actually be executed. The others become invalid. We will call these kind of requests causally related requests; they all have the same requestor. For the requestor it is important to know which of its request is executed, for the requests it is important to know of each others existence. For these purposes the requests are provided with two additional variables: a number and a reference to another request. All requests of the requestor are numbered uniquely, and the request that is actually being executed returns its number to the requestor. This is done by adding its number to the mailBox of the requestor. The mailBox of a process is a tool which can be used by other objects to pass information to the process. It is discussed in section 3.7.1. The reference to the next requests of the requestor is used to create a circle of causally related requests; the first request refers to the second request, the second to the third and so on. The last request refers to the first request again. See figure 3.3. The reference to another request can be used to invalidate the requests that are not selected. The selected request will send a message to its neighbour to invalidate itself, which will respond to it and pass the message through to his neighbour. In this way the invalidation message travels along all the requests in the circle of causally related requests. At some point the selected request will be reached, indicating that all causally related requests are invalidated. In this way the causally related requests can invalidate each other, and the result is that only one request is selected. See figure 3.4.
Figure 3.3 Circle of causally related requests.

Figure 3.4 Invalidation of causally related requests.
3.5 Guarded requests

POOSL process objects are allowed to perform guarded command statements. Guarded commands are statements preceded by an expression, called the guard of the statement. When a guarded command is executed the guard is evaluated. If it evaluates to true, the statement following the guard can be executed. See chapter 5 for a detailed discussion of this statement.

Every request allowed in POOSL is a candidate to be used in guarded command statements. To use the same requests that are used for unguarded requests, a mechanism is needed to distinguish unguarded request from guarded requests. Guarded requests represent guarded command statements and unguarded requests do not. For this purpose the classes representing a request are provided with an instance variable referring to the guard of the guarded command statement.

3.6 Statement request

The necessity for a statement request originates from the select-statement, see chapter 5. During the conversion of these statements into Smalltalk equivalents, we ran into the problem that an alternative could be any statement allowed in POOSL. This is illustrated with the following example of a select-statement.

```
sel
  if a > b
    then ch1!m1(a,b)
    else ch1!m1(b,a)
  fi
or
  a := a + 1
or
  methodCall(a,b)(returnValue)
or
  ch1!m1(a,b)
or
  ch1?m1(a,b)
les
```

The example shows a variety of possible alternatives. Conform the specification, all alterna-
tives must be left open until one is selected to be executed. The scheduler is used as selection mechanism. The advantage of using the scheduler for this purpose is that all alternatives are left open until the scheduler selects one.

For each alternative a request must be sent to the scheduler. For the alternatives representing a *message-send* or a *message-receive* statement a send or receive request can be used. To represent the other alternatives a similar request is implemented. Such a request is called a statement request. The statement of the alternative itself is not a part of the statement request. The link between a statement request and the corresponding statement is the number of the statement request.

If a request is executed it puts its number in the *mailBox* of the requestor. Next, it wakes up the requestor by sending a *signal* to its *semaphore*. If the requestor resumes its execution, it can determine which of its request is executed by reading the number of the request put in its *mailBox*. If the number of the executed request belongs to a statement request, the corresponding statement will be executed.

3.7 Modelling of process objects

The mapping of POOSL onto Smalltalk has consequences for the modelling of POOSL processes. In Smalltalk they are represented as objects and corresponding Smalltalk processes. These objects needs a mechanism enabling them to change their *simulation execution state* from *active* to *idle* and vice versa. Further, a *mailBox* is needed to receive data and to receive the number of the executed requests.

3.7.1 Process mailBox

Each process is provided with a *mailBox* which can be used to receive messages. The requests performed by a process use the *mailBox* to pass their number to the requestor, if they are executed. In this way the requesting process can find out which of its requests is executed in case it performed more than one request. The *mailBox* is also used to receive data. When a send request is executed, the data involved is added to the *mailBox* of the receiving process. If the receiving process becomes active again, it can read the data and the number of the request executed from its *mailBox*.

3.7.2 Simulator execution states

Another aspect of communication between processes are the *simulator execution states* of process objects. Communications in POOSL are synchronous, implying that the process can not proceed its execution until the execution of a *message-send* and corresponding *message-receive* statement is completed. In such a case the processes become *idle*. In the Smalltalk environment we will indicate the status of a process with the *simulator execution states*. To implement the *simulator execution states* transitions, we must provide
The Smalltalk equivalent of a POOSL process with an additional tool. For this purpose it is provided with a semaphore. If a process executes a communication or wait statement, it activates its semaphore. The simulator execution states of the process will change from active to idle. If the corresponding request is executed, it sends a signal to the semaphore of the requesting process. The result of this signal is that the simulation execution state of the requesting process changes from idle to active; the process can continue with its execution. A process has also two other states which are trivial: Born and Dead. Born indicate the creation of a process and Dead the end of its execution. The simulator execution states transition is given in figure 3.5.

**Figure 3.5 Simulation execution states transition.**
Chapter 4

Implementation of the simulator

The simulator is implemented in Smalltalk/V for Windows, a completely object-oriented and interactive programming environment. Everything used to implement the simulator is implemented in terms of objects and classes. In this chapter the implementation is discussed in detail. First the class hierarchy of the simulator is discussed, followed by a discussion regarding the implementation of the communication between processes. Aspects concerning the use of the simulator, such as input and output of the simulator are also subject of this chapter.

In figure 4.1 the scheme of the POOSL simulator is given. In this scheme the association between the different aspects of our design and implementation are visualized. Throughout this chapter references to this figure will be used to clarify the association of the subjects discussed and their place in the scheme presented in this figure.

![Diagram of POOSL simulator](image)

Figure 4.1 Scheme of the POOSL simulator.

4.1 The POOSL class

In figure 4.1 the scheme of the POOSL simulator is given. The implementation presented in this thesis is a part of the kernel. The kernel is the part of the simulator in which the actual simulation is performed. The implementation used to simulate the communication between POOSL process objects is an important part of the kernel.
To implement the concepts discussed in the previous chapter, a class hierarchy for the POOSL simulator is designed. All classes needed for the implementation are subclasses of the class POOSL. The class POOSL on its turn is a subclass of the class Object. The classes implemented until now, are classes concerning communication, processes and the implementation of the scheduler. Beneath the hierarchy of the class POOSL is shown. In the following subsections the functionality of the classes in this hierarchy are discussed. In Appendix A the protocol summary of each class is given.

Object
  POOSL
    PChannel
    PData
    PProcess
    PRequest
      PReceiveRequest
      PSendRequest
      PComRequest
      PWaitRequest
      PScheduledComReq
      PStatementRequest
      PScheduler
4.1.1 Class PChannel

This class represents a communication channel in the POOSL domain. An instance of this class is created for each channel used in a POOSL description. Its main task is to store incoming requests and to search for matching requests. If a match is found, an instance of the class PComRequest is created, initialized and added to the set requestsToSchedule of the scheduler. The scheduler is an instance of the class PScheduler.

If a process wants to communicate, it sends a request to the appropriate instance of this class. Two instance variables are used to store incoming requests: sendRequests and receiveRequests. Each time a channel receives a request, the invalid requests in these sets are removed. A channel can determine to remove a request by checking whether that request is valid or not.

The instance variables of PChannel are defined:

- receiveRequests : An instance of Set to collect all incoming receive requests.
- sendRequests : An instance of Set to collect all incoming send requests.
- name : An instance of String, indicating the channel's name.
- transTime : An instance of Number indicating the time needed to send a message through the channel.
- free : An instance of Boolean to indicate whether a channel is free.

4.1.2 Class PData

The class PData is used to subclass Smalltalk equivalents of POOSL data objects. It is an empty class; it does not contain any instance variables or methods.

4.1.3 Class PProcess

The class PProcess is used to subclass Smalltalk equivalents of POOSL process classes. It contains a number of instance methods and instance variables which will be inherited by all subclasses of this class. In this class the mailbox and the semaphore of the processes, which were introduced in chapter 3, are defined. It also contains a method to simulate tail recursion, which will be discussed in section 5.1.3.

4.1.4 Class PRequest

The class PRequest is a metaclass. It is used as an abstract class to group all possible requests that are supported. It does not contain any instance methods or variables. In the following subsections its subclasses are discussed.
4.1.4.1 Class PComRequest

PComRequest represents a communication between two processes. It contains references to the matching send and receive request. An instance of this class is created by an instance of the class PChannel and is added to the set requestsToSchedule of the scheduler, an instance of PScheduler.

An instance of PComRequest is executed if it receives the message execute from the scheduler. After receiving this message an instance of PscheduledComReq is created and added to the list scheduledRequests of the scheduler. Before this can be done, some checks have to be performed to determine whether the communication can take place or not.

The instance variables used in PComRequest to store the information needed are:

- **sendRequest**: A reference to an instance of PSendRequest, i.e. the send request belonging to this communication request.
- **receiveRequest**: A reference to an instance of PReceiveRequest, i.e. the receive request belonging to this communication request.
- **channel**: A reference to an instance of PChannel, i.e. the channel used for communication.
- **valid**: An instance of Boolean indicating whether this communication request is still valid or not.

4.1.4.2 Class PWaitRequest

PWaitRequest represents the request of a process to delay its execution for some time. Time requests can be used to simulate the real time aspects of POOSL specifications. An instance of this class is created, initialized and added to the set scheduledRequests of the scheduler by the requesting process. The delay corresponding to the wait request is used to schedule the wait request. Suppose a process wants to perform a wait request of two simulation steps. To do this it sends the following message to the scheduler: PScheduler active scheduleRequest:waitRequest after:delay. waitRequest is an instance of PWaitRequest and delay is time corresponding to the wait request. An instance of PWaitRequest is executed when it receives the message execute from the scheduler.

The information needed are stored in the instance variables:

- **requestor**: A reference to an instance of PProcess which is the requestor.
- **valid**: An instance of Boolean indicating whether this wait request is still valid.
- **nextRequest**: A reference to an instance of a subclass of PRequest representing the next request of the requestor.
- **number**: An instance of Number indicating the number of this request.
4.1.4.3 Class PSendRequest

*PSendRequest* represents the request of a process to send a message. An instance of this class is created and initialized by the requesting process. It is sent to the channel used for communication. An instance of this class can be used as a guarded as well as an unguarded send request. The instance variable *guardExpression* is used to distinguish guarded request from unguarded requests. An instance of this class is executed if it receives the message *execute* from an instance of the class *PScheduledComReq*.

The information needed is stored in the instance variables:

- **name**: An instance of *String* indicating the message name of the send request.
- **requestor**: A reference to an instance of *PProcess* indicating the requestor of this send request.
- **guardExpression**: A reference to a block of code representing the guard of a guarded request.
- **receiver**: A reference to an instance of a subclass of *PProcess* indicating the receiver of this send request.
- **valid**: An instance of *Boolean* indicating whether this request is still valid.
- **nextRequest**: A reference to an instance of a subclass of *PRequest* referring to the next request of the requestor.
- **number**: An instance of *Number* indicating the number of this request.
- **channel**: A reference to an instance of *PChannel* referring to the channel used to send the message.
- **arguments**: An instance of *Collection* containing the possible data to send.

4.1.4.4 Class PReceiveRequest

*PReceiveRequest* represents a request of a process to receive a message. An instance of this class is created and initialized by the requesting process. After creation and initialization it is sent to the channel used for communication. An instance of this class can be used as guarded as well as unguarded receive request. The instance variable *guardExpression* is used to distinguish guarded request from unguarded requests.

An instance of this class is executed when it receives the message *execute* from an instance of the class *PScheduledComReq*.

The instance variables defined in the class *PReceiveRequest* are:

- **name**: An instance of *String* indicating the message name.
- **guardExpression**: A reference to a block of code representing the guard of a guarded command.
- **requestor**: A reference to an instance of a subclass of *PProcess* referring to the requesting process.
- **valid**: An instance of *Boolean* indicating the validity of the request.
- **nextRequest**: A reference to an instance of a subclass of *PRequest* referring to the
Implementation of the simulator

next request of the requestor.

channel : A reference to an instance of PChannel referring to the channel used
to receive the message.

condition : A reference to a block of code representing the condition for commun-

4.1.4.5 Class PScheduledComReq

This class is used to schedule a communication request which is actually going to be exe-
cuted. It is responsible for the execution of the send and receive request in the right order.
If this was not controlled the possibility could exist that an instance of PReceiveRequest is
executed before the matching instance of PSendRequest is executed. In that case the re-
questor of the receive request would try to read the data from its mailBox which is not
present, because the send request is not executed yet. This would result in an error.
An instance of this class is created by an instance of the class PComRequest and added to the
set scheduledRequests of the scheduler. It is executed if it receives the message execute from
the scheduler.

The information needed is stored in instance variables:

requestToSend : A reference to an instance of the class PSendRequest referring to the send
request.

nextRequest : A reference to an instance of the class PReceiveRequest referring to the
receive request.

4.1.4.6 Class PStatementRequest

PStatementRequest represents a statement request of a process. The requesting process
creates an instance of this class, initializes it and adds it to the set requestsToSchedule of the
scheduler. If an instance of this class receives the message execute from the scheduler it is
executed. Because this request is not time related it is not necessary to add this request or a
representative of it to the set scheduledRequests of the scheduler. Once it receives the mes-

age execute, its task is to invalidate the causally related requests and wake up the requestor.

The instance variables defined in the class PStatementRequest are:

name : An instance of String holding the name of the statement request.

guardExpression : A reference to a block of code representing the guard of a guarded
command.

requestor : A reference to an instance of a subclass of PProcess referring to the
requesting process.

valid : An instance of Boolean indicating the validity of the request.

nextRequest : A reference to an instance of a subclass of PRequest referring to the
next request performed by the requestor.
4.1.5 Class PScheduler

An instance of the class PScheduler is used to represent the scheduler of the simulator. It is used to model and control the time and communications in the Smalltalk environment. The scheduler is also responsible for the progress of the simulation. It steps through the simulation, increasing the simulation time by each step it makes. At each step the requests in scheduledRequests and requestsToSchedule, that can be handled at that simulation time, are handled.

A simulation session is started by sending the message new initialize activate startUp to the class PScheduler. Once the simulation is started it is controlled by the instance method step of the class PScheduler. The execution of the scheduler is ended if there are no more requests left in both the set scheduledRequests and the set requestsToSchedule.

Class PScheduler has three instance variables:
requestsToSchedule : An instance of Set to collect incoming requests and requests which has to be scheduled.
scheduledRequests : An instance of PRequestsList to hold requests which are scheduled.
time : An instance of Number indicating the current simulation time. It is initialized to have value zero.

4.2 Handling communication requests

In this section the implementation of the conceptual communication design introduced in section 3.2.1, is discussed. A detailed discussion involving implementation aspects is given.

If a Smalltalk equivalent of a POOSL process wants to communicate with an other process it performs a request to send or a request to receive a message. To do such a request it creates an instance of the class PSendRequest or PReceiveRequest, depending on the kind of request it wants to perform. Next, the parameters required for the execution of the request are passed to it. The requesting process has to initialize the name of the request, the channel used for communication, the number of the request, and the process object itself is defined as the requestor of the request. In case of a send request the data is also passed to the request. If the requests is used for a conditional receive request a reference to the block implementing the condition is passed to the request. After initialization the request is sent to the channel that is used for communication. Channels are instances of the class PChannel. Sending a request to a channel is done by sending the message receive:aPReceiveRequest or send:aPSendRequest to the channel. If a receive request is sent to the channel, the first message is sent to the channel. In case of a send request, the second message is sent to the channel. The channel will add the received request to sendRequests or receiveRequests. See figure 4.2.

Next, the channel searches for requests that match with the newly received request. If a
Implementation of the simulator

Figure 4.2 Processes performing requests to communicate.

send request is received, the channel searches in receiveRequests for candidates, otherwise a candidate is searched in sendRequests. The benefit to use two sets instead of one is that the search time is kept as short as possible.

For each match a communication request is created and initialized. This is done in the instance method pair: with: of the channel object. The matching send and receive requests are passed to the created communication request. After initialization, the communication request is added to the set requestsToSchedule of the scheduler by sending the message scheduleCom: aPComRequest to the scheduler. The scheduler is an instance of the class PScheduler. The creation of a communication request and sending it to scheduler is illustrated in figure 4.3.

Figure 4.3 Channel sending a communication request.
If the scheduler starts to trigger the requests stored in its instance variable `requestsToSchedule` the communication request added to this set will be triggered too. When the trigger is received by the communication request this request will check whether the communication can be executed. First, it will evaluate the block implementing the guard of both the receive and the send request. If this check is passed, the next check performed is the evaluation of the block implementing the condition (valid or invalid) of the receive request. If this check is successful too the communication request checks whether the channel used for communication is free. If one of these checks fails, the communication is not executed and will remain in `requestsToSchedule`. In case all checks are passed the communication will be executed. This is done by blocking the channel and invalidating causally related requests. An instance of `PScheduledComReq` is created and added to the set `scheduledRequests` of the scheduler. The communication request itself is made invalid meaning that it can be removed from `requestsToSchedule`. When the instance of `PScheduledComReq` receives the message `execute` from the scheduler it will pass this message first to the corresponding send request and then to the receive request. See figure 4.4.

![Figure 4.4 Execution of aPScheduledComReq.](image)

If the message `execute` is by the send and the receive requests, they are executed and the requestors of them are woken up. This is done by sending a signal to their semaphore. This completes the execution of a communication between two processes.
4.3 Simulator input

In figure 4.1 we see that the simulator must be able to read a POOSL specification before it can be simulated. The specification read by the simulator must contain all the information the simulator needs to simulate the design.

At this phase of the implementation the input of a system specification from a file or by other means was not a subject. The test designs we used were programmed directly in Smalltalk. What we have done regarding to the input for the simulator, is that we have described rules how to convert POOSL descriptions into Smalltalk equivalents. This is discussed in chapter 5. The output of the conversion can be used as input for the simulator.

A class can be read from a file and added to the existing class hierarchy of Smalltalk by evaluating the following Smalltalk code:

\[
\text{(File pathName: 'full path and filename')} \text{ fileIn; close}
\]

Execution of this code results in adding the classes defined in the inputfile to the existing Smalltalk class hierarchy. The place of the class in the Smalltalk class hierarchy, must be defined in the file. This information must be defined before the actually class definition is given. Suppose we want to read a inputfile with the following contents:

\[
\begin{align*}
\text{PProcess subclass: #PExampleClass} \\
\text{instanceVariableNames: ''} \\
\text{classListVariableNames: ''} \\
\text{poolDictionaries: '!' } \\
\text{PExampleClass class methods ! !} \\
\text{PExampleClass methods !} \\
\text{execute} \\
\text{"Print text to the Transcript of Small talk."} \\
\text{Transcript show: 'This file is an example to import a class in Small talk.'; cr;} \\
\text{show: 'The output of a parser-code-generator should look like this.'; cr !}
\end{align*}
\]

The first line defines the place and the name of the class followed by the class definition. It will be added as a subclass of the class PProcess and it will be named PExampleClass. The class PExampleClass contains only one instance method, namely execute. It has no instance variables, no class variables, no pooldictionaries or class methods. This is what the output of a possible parser-code-generator must look like if we want to use it as input of our simulator. All classes defined in a POOSL specification must be added as a subclass of the PProcess. The input of our simulator is further described in chapter 7.

4.4 Simulator output

Just like the simulator input, not much attention is paid to the output of the simulator at
Implementation of the simulator

during this phase of the implementation. However, the output will become an important part of our simulator in the future. See future research in chapter 7.

The output presented to the user will be generated by the kernel of the simulator. See figure 4.1. The POOSL specification which is used as input must not contain any output information. The output generated by the kernel and presented to the user will inform the user about the progress of the simulation. This can be done graphical, textual or a combination of both. The output of IDaSS can be used as an example for the simulation output.

The output of the simulator implemented so far is a simple textual output. The output is presented in the Transcript of Smalltalk. During simulation, text is written to it, indicating the simulation progress. The output consists of information such as the simulation time, the requests received by the channels, status of the channels, the requests executed, requests in the requestsToSchedule and schedulerRequests set and other information concerning the communication between processes. In appendix 3 an example of the output is given.
Chapter 5

Smalltalk equivalents for POOSL syntax

Once a POOSL description of a system is made, it must be mapped onto a Smalltalk equivalent. Although POOSL and Smalltalk have a lot in common, they cannot be mapped one-on-one. A conversion step has to be made to map a POOSL description onto a Smalltalk equivalent as described in the previous chapters. This is the subject of this chapter.

To convert the syntax of POOSL into Smalltalk equivalents conversion rules must be given. The syntax defined in POOSL can be divided into the following categories:

- Process statements
- Process classes
- Cluster classes
- Data classes

In the following subsections conversion rules of these categories are described. However, the conversion of the cluster classes is not described yet. This conversion is still subject of future research.

5.1 Process statements

Process statements are used in POOSL to specify the behaviour of process objects. A total of 10 process statements are defined. The set of process statements is defined as follows:

\[ SP ::= S \]
\[ ch!m(E_1,..,E_n) \]
\[ ch?m(p_1,..,p_n \mid E) \]
\[ S^p_1; S^p_2 \]
\[ m(E_1,..,E_n)(p_1,..,p_n) \]
\[ Sel S^p_1 \ OR \ S^p_2 \ Les \]
\[ \{E\} S^p \]
\[ if \ E \ then \ S^p_1 \ else \ S^p_2 \ fi \]
\[ While \ E \ do \ S^p \ od \]
\[ S^p_1 > > S^p_2 \]

The first statements \( S \) is a statement of data objects and will be discussed in section 5.3. In the following subsections the conversion rules of these process statements will be discussed. At the end of each subsection an example will be given to illustrate the conversion.
Due to the complexity of the *Interrupt-statement* and the *Disrupt-statement*, the conversion of this statement is not described yet and is subject of future research. The problem with this statement is that $S^p_2$ can act on variables which are not accessible for $S^p_2$ because the part of the stack used to store these variables cannot be accessed by $S^p_2$.

5.1.1 Message-send

With this statement a POOSL process is able to do a request to send a message to another POOSL process. The formal POOSL syntax for this statement is:

```plaintext
ch!m(X_1, ..., X_n)
```

with
- \( \text{ch} \) : The channel used for communication.
- \( m \) : The name of the message.
- \( X_1, ..., X_n \) : The data involved.

This statement is unknown in Smalltalk. Below a summary is given of the actions to take if we want to convert this POOSL statement into a Smalltalk equivalent.

- Create and initialize an array with the data to send. This array will be used to pass the data to the receiving process. This is only needed if data is sent.
- Create a new instance of `PSendRequest`. This is an object representing a send request. All relevant information needed for communication are initialized and passed to the send request. Initializing an instance of `PSendRequest` involves:
  - defining the requestor.
  - defining the name of the send request.
  - defining the channel used for communication.
  - defining the number of the send request.
  - defining the data to send.
- Define a reference to the next request of the requestor.
- Send the send request to the channel involved.
- Activate the process `semaphore`.
- Remove the number of the request executed from the `mailbox` of the process if the process is activated again.

The result is the following Smalltalk equivalent:

```smalltalk
| sendRequest data requestExecuted |
data: = Array new:n.
data at:1 put:X1.
data at:2 put:X2.
...
data at:n put:Xn.
```
POOSL description of a method:
  example _sendRequest
      | channel1!message1(1,2,3)

Smalltalk Equivalent:
  example _sendRequest
      | sendRequest data requestExecuted|
      data: = Array new:3.
      data at:1 put:1.
      data at:2 put:2.
      data at:3 put:3.
      sendRequest: = PSendRequest new initialize; channel: channel1;
      name: 'message1'; number: 1; arguments: data; requestor: self.
      sendRequest nextRequest: sendRequest.
      channel1 send: sendRequest.
      self wait.
      requestExecuted: = mailBox removeFirst.

Remark: Notice that the neighbour of the request is the request itself!
5.1.2 Message-receive

With this statement a POOSL process is able to do a request to receive a message from another POOSL process. The formal POOSL syntax for this statement is:

```
ch?m(p₁, ... ,pₙ|E)
```

- `ch` : The channel used for communication.
- `m` : The name of the message.
- `X₁, ..., Xₙ` : The data involved.
- `E` : The condition for communication.

This statement is almost similar to the `message-send` statement described in the previous section. Below a summary is given of the actions to take if we want to convert a `message-receive` into a Smalltalk equivalent.

- Create a block of code for the implementation of the condition. This is only necessary in case of a conditional receive request.
- Create a new instance of `PReceiveRequest`. This is an object representing a receive request. The relevant information needed for the communications involves:
  - defining the requestor.
  - defining the name of the request.
  - defining the channel used for communication.
  - defining the number of the request.
  - defining the condition for communication.
  - defining references to the next requests of the requestor.
- Send the receive request to the channel involved.
- Activate the process `semaphore`.
- Remove the data from the `mailBox`.
- Remove the number of the request executed from the `mailBox` of the process if the process is activated again.

The data received is written directly in the `mailBox` of the requestor. This is done by the corresponding send request. As a result of this the receive request does not have to concern about the data received. The requesting process itself is responsible for the handling of the data received. The number of the request is also removed from the `mailBox` to empty it.

The result is the following Smalltalk equivalent:

```smalltalk
| receiveRequest expression data requestExecuted |
expression:=[:ar | E ].
```
Smalltalk equivalents for POOSL syntax

receiveRequest:= PReceiveRequest new initialize; name:'m'; channel:ch; number:aNumber; requestor:self; condition:expression.
receiveRequest nextRequest:aPRequest.
ch receive:receiveRequest.
self wait.
data:= mailBox removeFirst.
requestExecuted:= mailBox removeFirst.

With 
- receiveRequest : An instance of PReceiveRequest referring to the request.
- data : An instance of Array used to remove data from mailBox.
- ch : An instance of PChannel, the channel used for communication.
- 'm' : An instance of String indicating the name assigned to request.
- aNumber : An instance of Number, indicating the number of this receive request.
- aPRequest : An instance of a subclass of PProcess reference to the next request of the requestor.
- ar : An instance of Array used to pass the data which will be received to the block which is used to implement the condition of the message-receive statement. It is possible that the data which will be received is needed to evaluate E.

To illustrate the modelling of a POOSL message-receive statement into its Smalltalk equivalent an example is given.

POOSL description of a method:

```
exampLe _receiveRequest
| a b c |
channel?message(a,b,c | a+b+c > 3).
c:= a + b.
```

Smalltalk Equivalent:

```
exampLe _receiveRequest
| a b c receiveRequest expression data requestExecuted |
expression:=[ar | a:=ar at:1. b:=ar at:2. c:=ar at:3. a+b+c > 3].
receiveRequest:= PReceiveRequest new initialize; channel:channel;
name:'message'; number:-1; requestor:self; condition:expression.
receiveRequest nextRequest:receiveRequest.
channel receive:receiveRequest.
self wait.
data:= mailBox removeFirst.
RequestExecuted:= mailBox removeFirst.
a:=data at:1.
b:=data at:2.
c:= a + b.
```
Remark: Notice that the neighbour of the request is the request itself!

5.1.3 method-call

By means of such a method-call statement a process object can call one of its methods. The formal POOSL syntax for this statement is:

\[
m(E_1, \ldots, E_n)(p_1, \ldots, p_m)
\]

with

\[m:\] Name of the method called.
\[E_i, \ldots, E_n:\] Input parameters.
\[p_1, \ldots, p_n:\] Output parameters.

If a number of arguments has to be passed, the syntax used for calling a method in Smalltalk is not practical. It will result in unnecessary long message patterns. A more practical way to pass the arguments is to use an array. All arguments are put in an array which will be passed to the method.

A Smalltalk method can only return one object as a result. If a POOSL method returns more than one parameter we have to find a way to realize this. The solution is to use an array. The method puts the parameters it wants to return in an array and returns the array to the caller.

Using this approach, the following Smalltalk equivalent is defined:

```smalltalk
| data |
data := Array new:n.
data at:1 put:x1.
...
data at:n put:xn.
data := self perform:methodName with:data.
```

`self` is a reserved word in Smalltalk. With this special variable objects are able to send messages to themselves. This makes it possible for an object to call one of its own methods. `perform:with:` and `perform:` are methods of the class `Object`, which can be used to call a method of a class. The argument this methods require are the name of the method called and the argument passed to it. If no arguments has to be passed, `perform:` is used, otherwise `perform:with:` is used.

It is possible to use tailrecursion in POOSL description. Tailrecursion means that a method calls itself at the end of its execution. Further no parameters are returned, which means that \( m = 0 \). To prevent stack overflow a special method is supplied which can be used to simulate tailrecursion. It is the instance method `executeTailRecursion:aMessage` of the class `PProcess`. `aMessage` is an instance of the class `Message`. This message contains a selector representing the method which has to be executed recursively. This selector must be defined by the object calling `executeTailRecursion`. In `executeTailRecursion: the message perform:` is used in a while-loop to simulate tailrecursion.
Smalltalk equivalents for POOSL syntax

If we want to use `executeTailRecursion` to simulate recursive execution of a method, the method specification must be changed. The message used for the recursion must be replaced with a statement returning the name of the method to the receiver. This will be an indication for `executeTailRecursion` to continue the while-loop. The while-loop will be ended if the method called does not return its name. Each message used to call a recursive method must be replaced with a message to call `executeTailRecursion`.

Following an example is given how to convert a method call in POOSL into its Smalltalk equivalent.

**POOSL description of a method:**

```
example_Method_call
| e1 e2 e3 p1 p2 |
methodCalled(e1,e2,e3)(p1,p2)
```

**Smalltalk equivalent:**

```
example_method_call
|dataArray e1 e2 e3 p1 p2 |
dataArray:=Array new:3.
dataArray at:1 put:e1.
dataArray at:2 put:e2.
dataArray at:3 put:e3.
p1:=dataArray at:1.
p2:=dataArray at:2.
```

5.1.4 sequential composition

With this statement a POOSL process is able to define two process statements which has to be executed in a sequence. The sequence in which the statements has to be executed is the same in which they appear in the sequential composition statement.

The formal POOSL syntax for this statement is:

```
S^p_1;S^p_2
```

with

```
S^p_1 : Process statement 1.
S^p_2 : Process statement 2.
```

The Smalltalk equivalent of this statement straightforward. It is composed with the Smalltalk equivalent of the first statement followed by the Smalltalk equivalent of the second statement.

To illustrate the conversion of sequential composition statement into its Smalltalk equiva-
lent the next example is given.

**POOSL description of a method:**

```
example_sequence
  | a b |
  ch1?m1(a b);
  a:=a+b; b:=5;
  ch2!m2(a b)
```

**Smalltalk equivalent:**

```
example_sequence
  | receiveRequest sendRequest a b requestExecuted data |
  receiveRequest:=PReceiveRequest new initialize; channel:ch1;
    name:'m1'; number:=1; requestor:self.
  receiveRequest nextRequest:=receiveRequest.
  ch1 receive:=receiveRequest.
  self wait.
  data:= mailBox removeFirst.
  requestExecuted:= mailBox removeFirst.
  a:=data at:1.
  b:=data at:2.
  a:=a+b.
  b:=5.
  data at:1 put:a.
  data at:2 put:b.
  sendRequest:=PSendRequest new initialize; channel:ch2;
    name:'m2'; number:2; arguments:data; requestor:self.
  sendRequest nextRequest:=sendRequest.
  ch2 send:=sendRequest.
  self wait.
  requestExecuted:= mailBox removeFirst.
```

### 5.1.5 select statement

The *select-statement* is used to indicate that a process can choose between different alternatives. The formal POOSL syntax for this statement is:

```
sel S^1 or .. or S^n les
```

with \( S^i \): A process statement.

This statement needs special attention because each alternative can be every statement allowed in POOSL. In section 3.6 the conceptual solutions for the representation and handling of the alternatives are given.
Below the resulting Smalltalk equivalent of the select-statement is given. Since it is impossible to give an exact description of the conversion rule a general description is given how to translate a select-statement into a Smalltalk equivalent. The difficulty is that the number of alternatives may vary from case to case. The same holds for the statements that are used as alternatives. In the following description at some places plain text is placed between quotes to indicate the actions that must be taken.

| requestExecuted ... |
"Send a requests for each alternative to the scheduler"

self wait.
requestExecuted := mailBox removeFirst.
"If requestExecuted belongs to a receive request, remove the data from the mailBox."
"If requestExecuted belongs to a statement request, execute the corresponding statement."

Below an example is given to convert a select-statement into its Smalltalk equivalent.

**POOSL description of a method:**

```plaintext
example_select
  | a b c |
a := 1; b := 2; c := 3;
sel
  channel1!message1(a,b,c)
  or
  channel1?message2(a,b,c)
  or
  a := a + b
les
```

**Smalltalk Equivalent:**

```plaintext
example_select
  | sendRequest receiveRequest data requestExecuted a b c |
a := 1. b := 2. c := 3.
data := Array new:3.
data at: 1 put: a.
data at: 2 put: b.
data at: 3 put: c.
sendRequest := PSendRequest new initialize; channel: channel1;
  name: 'message1'; arguments: data; number: 1; requestor: self.
receiveRequest := PReceiveRequest new initialize; channel: channel1;
  name: 'message2'; number: 2; requestor: self.
statementRequest := PStatementRequest new initialize; name: 'aName';
  channel: statementChannel; number: 3; requestor: self.
sendRequest nextRequest: receiveRequest.
receiveRequest nextRequest: statementRequest.
```
Smalltalk equivalents for POOSL syntax

statementRequest nextRequest:sendRequest.
channel1 send:sendRequest.
channel1 receive:receiveRequest.
PScheduler active scheduleCom:statementRequest.
sel wait.
requestExecuted:=mailBox removeFirst.
(choice > 1)
  ifTrue:
    (choice=2)
      ifTrue: [ data:=mailBox removeFirst.
        a:=data at:1.
        b:=data at:2.
        c:=data at:3]
      ifFalse:[ a:=a+b+c]
    ]

Remarks: - Notice the way the neighbours are defined!
- Notice that the statement request is sent directly to the scheduler.
- If the process is woken up, requestsExecuted is used to determine which of the requests is actually executed. If the receive request is executed, the data received are removed from the mailBox. If the statementRequest is executed, the corresponding statement is executed.

5.1.6 if-statement

The if-statement is used as the usual if-statement. The formal POOSL syntax of this statement is:

\[
\text{if } E \text{ then } S_P^1 \text{ else } S_P^2 \text{ fi}
\]

with \( E \): An expression.
\( S_P^1 \): An process statement.

If the expression \( E \) evaluates true, the statement \( S_P^1 \) is executed. If the expression evaluate false \( S_P^2 \) will be executed.

If-statements are also defined in Smalltalk. The only different is that Smalltalk uses a different syntax. The Smalltalk equivalent of the if-statement is:

\[
(E)
\text{ifTrue: } [S_P^1]
\text{ifFalse: } [S_P^2].
\]

This makes converting the if-statement of POOSL into its Smalltalk equivalent not so
Smalltalk equivalents for POOSL syntax

difficult. The most effort is needed to replace $S_1$ and $S_2$ with their Smalltalk equivalents.

An example is given to illustrate the converting of an if-statement.

**POOSL description of a method:**

evaluate

```
if a+b > 1
  then channel1!message1(a b)
  else channel2!message2(a b)
fi
```

**Smalltalk equivalent:**

evaluate

```
(a+b > 1)
ifTrue: [:sendRequest :data |
  data:=Array new:2.
  data at:1 put:a.
  data at:2 put:b.
  sendRequest:=PSendRequest new initialize; channel:channel1;
  name: 'message1'; number:1; arguments:data.
  sendRequest nextRequest:sendRequest.
  channel1 send:sendRequest.
  self wait]
ifFalse: [:sendRequest :data |
  data:=Array new:2.
  data at:1 put:a.
  data at:2 put:b.
  sendRequest:=PSendRequest new initialize; channel:channel2;
  name: 'message2'; number:2; arguments:data.
  sendRequest nextRequest:sendRequest.
  channel1 send:sendRequest.
  self wait].
```

### 5.1.7 guarded command

If a process statement is proceeded with an expression $E$, it is called a guarded command.

The formal POOSL syntax for a guarded command statement is:

\[
[E] \ S^p
\]

with $[E]$ : The guard, an expression, of the statement.

$S^p$ : A process statement.
This command is used in case the execution of a process statement depends on the variable state of the process. If the guard evaluates to true and the statement following the guard can be executed, the guarded command statement will be executed. In all other cases it will not be executed. For example, if the statement following the guard is a request to send a message, it is possible that the request can not be executed at once. The reason can be that the collaborator is not ready to communicate.

The Smalltalk equivalent of this statement is quite straightforward. For each guarded command statement a request is sent to the scheduler. At this point we are confronted with the same problem as with the select-statement. The statement following the guard can be any statement. This problem is handled in the similar way as it is handled in case of the select-statement. See sections 3.6, 3.6 and 5.1.6.

Below the rules for the resulting Smalltalk equivalent of the guarded command statement is given. Since it is impossible to give an exact description of the conversion rule, a general description is given. The difficulty is that the number of alternatives may vary from case to case. The same holds for the statements that are used as alternatives. In the description at some places plain text is place between quotes to indicate the actions that must be taken at that place.

| guard requestExecuted .... | guard:= aBlock.  
"Perform a request representing SP with guardExpression referring to guard"  
self wait.  
requestExecuted:= mailBox removeFirst.  
"If a receive request is executed, remove the possible received data from the mailBox.  
If a statement request is executed, execute the corresponding statement"

Depending on the number of guarded commands used, a Smalltalk equivalent can become quite extensive.

The following example is almost the same example used in section. The difference between this example and the example in section is that the select-statement is replaced with guarded commands. In the following example a process can choose between three alternatives, an alternative to execute a normal statement, one to send a message or one to receive a message. Whether an alternative is valid, depends on the guard of the alternative.

**POOSL description of a method:**
```
example_guarded_command
| a b c |
\texttt{a:=1; b:=2; c:=3;}  
\texttt{[a+b+c > 6] a:=a+b}  
\texttt{[a>0] channel1!message1(a,b,c)}  
\texttt{[a<0] channel1?message2(a,b,c)}
```

\[41\]
Smalltalk equivalents for P00SI syntax

Smalltalk Equivalent:

``` Smalltalk 

sendRequest receiveRequest statementRequest data choice a b 
expression1 expression2 expression3 | 
  a:=1. b:=2. c:=3. 
  expression1:=[a > 0]. 
  sendRequest:=PSendRequest new initialize; channel:channel1; name: 'message1'; 
    arguments:data; number:1; guardExpression:expression1; requestor:self. 
  expression2:=[a < 0]. 
  receiveRequest:=PReceiveRequest new initialize; channel:channel1; 
    requestor:self; name: 'message2'; number:2; guardExpression:expression2. 
  expression3:=[a + b + c > 6]. 
  statementRequest:=PStatementRequest new initialize; name: 'r1'; requestor:self; 
    number:3; guardExpression:=expression3. 
  sendRequest nextRequest:receiveRequest. 
  receiveRequest nextRequest:statementRequest. 
  statementRequest nextRequest:sendRequest. 
  channel1 send:sendRequest. 
  channel1 receive:receiveRequest. 
  PScheduler active scheduleCom:statementRequest. 
  self wait. 
  choice:=mailbox removeFirst. 
  (choice > 1) 
    ifTrue: [ 
      (choice=2) 
        ifTrue: [ data:=mailbox removeFirst. 
          a:=data at:1. 
          b:=data at:2. 
          c:=data at:3] 
        ifFalse: [a:=a+b+c] 
      ] 
```

Remarks:
- Notice the way the neighbours are defined!
- If the process is woken up, choice is used to determine which of the requests is actually executed. If the receive request is executed, the data received are removed from the mailbox. If the statement request is executed, the corresponding statement is executed.

5.1.8 while-statement

The while-statement is used as the usual while-statement. As long as the expression E evaluates to true the statement S will be executed.
The formal POOSL syntax for this statements is:

```
while E do S' od
```

with \( E \) : An expression.
\( S' \) : A process statement.

Smalltalk itself supports a *while-statement*. Converting the POOSL version into a Smalltalk equivalent is a straightforward conversion. The only thing we have to do is to fit the POOSL *while-statement* into the Smalltalk representation of the *while-statement*. The Smalltalk syntax of the *while-statement* is as follows:

```
[E] whileTrue: [S'].
```

The process statement \( S' \) is replaced with its Smalltalk equivalent. This is illustrated in the following example.

**POOSL description of a method:**
```
example_while
    | a |
    a:=1;
    while [a < 10] do a:=a+1 od
```

**Smalltalk equivalent:**
```
example_while
    | a |
    a:=1.
    [a < 10]
    whileTrue: [a:=a+1].
```

### 5.2 Process class

Within a process class definition the functionality of the instances of the class is specified.

The POOSL syntax of the process class is defined as follows:

- **process class** : Name and external variables.
- **instance variables** : Set of all instance variables.
- **communication channels** : All channels used for communication.
**Smalltalk equivalents for POOSL syntax**

- **message interface**: List of abstract send or receive actions.
- **initial method call**: Method to start the process activities.
- **instance methods**: A number of method definitions.

This specification must be mapped onto the Smalltalk equivalent given in section 2.2. Not all the items of the POOSL class definition are used for the mapping. The message interface is not needed. The communication channels are represented by instance variables.

The instance methods are converted into Smalltalk equivalents using the conversion rules described in the previous section. The instance methods in the POOSL specification specify the behaviour of the process class. They are not supposed to generate output to the users interface. The output displayed in the users interface must be generated by the kernel of the simulator. At this phase the kernel is formed by the class `PRequest` and its subclasses, the class `PChannel` and the class `PScheduler`. The *Transcript* of Smalltalk is used as users interface. The output displayed in the Transcript are mostly information about the requests managed by the channels and the scheduler. See section 4.4.

The initial method call is used to start up the process. This method will be called in the method used to initialize the simulation environment of the design. At this phase of the implementation, the method `startUp` of the class `PScheduler` is used to initialize and start up the simulation environment of the design.

The following example gives the Smalltalk equivalent of a process class defined in POOSL. The resulting Smalltalk equivalent is supposed to be the contents of a file, in which it is stored.

**POOSL description:**

```
process class
ExampleProcess

instance variables
a, b

communication channels
input output

message interface
input?received(a,b) output?sent(a,b)

initial method call
configureAndStart

instance methods
```
configureAndStart
"Change the data received and send them away"
| temporary |
input?received(a,b);
temporary:=a;
a:=b; b:=temporary;
output?sent(a,b);

Smalltalk equivalent:

\begin{verbatim}
POProcess subclass: #ExampleProcess

instanceVariableNames: 'a b input output'
classVariableNames: ''
poolDictionaries: ''

ExampleProcess class methods

ExampleProcess methods

configureAndStart
"Change the data received and send them away"
| temporary sendRequest receiveRequest requestExecuted data |
receiveRequest:=PReceiveRequest new initialize; channel:input;
name: 'received'; number: 1; requestor:self.
receiveRequest nextRequest:receiveRequest.
input receive:receiveRequest.
self wait.
data:= mailBox removeFirst.
requestExecuted:= mailBox removeFirst.
a:=data at:1.
b:=data at:2.
temporary:=a.
A:=b; b:=temporary.
anArray:=Array new:2.
anArray at:1 put:a.
anArray at:2 put:b.
sendRequest:=PSendRequest new initialize; channel:output;
name: 'sent'; number:2; arguments:anArray; requestor:self.
sendRequest nextRequest:sendRequest.
output send:sendRequest.
self wait.
requestExecuted:= mailBox removeFirst!!
\end{verbatim}

5.3 Data objects

The data objects of POOSL are much alike objects in Smalltalk. Due to this similarity the conversion of data objects into Smalltalk equivalents is not a problem. The statements used
Smalltalk equivalents for POOSL syntax

are almost similar to the statements of Smalltalk. However, some statements may differ from the Smalltalk protocol. The data statements of POOSL are given by:

\[
\begin{align*}
S := & \ E & \text{: An expression.} \\
X := & \ E & \text{: Assignment to an instance variable.} \\
U := & \ E & \text{: Assignment to a local variable or parameter.} \\
S_1;S_2 & \text{: Sequential composition.} \\
\text{if } E \text{ then } S_1 \text{ else } S_2 \text{ fi} & \text{: If-statement.} \\
\text{while } E \text{ do } S \text{ od} & \text{: While-statement.}
\end{align*}
\]

None of these statements are new to us. Assignments are trivial statements. The other statements, sequential composition, if-statement and while-statement are discussed in the subsections of section 5.1.
Chapter 6

Test case

6.1 Specification

To test our simulator we designed it was used to test a number of cases. One of these designs will be discussed here. The design used is shown in figure 6.1.

![Test design](image)

*Figure 6.1 Test design.*

This design is used to simulate a *select-statement*. Our goal is to investigate the non-deterministic behaviour of the selecting mechanism, built in the simulator. The design used for the test consists of four processes: process A, process B, process C and process D. Process C wants to receive a message from process A or process D. However, process C is only willing to communicate within a limited time interval. If none of the processes A or B are ready to communicate with process C within this time interval, process C sends a message *TimeOut* to process D. If process C did communicate, it sends the name of the collaborator and the data received to process D.

To illustrate the non-deterministic behaviour of the selecting mechanism, the execution of these processes are recursive.

The POOSL specification and the Smalltalk equivalents of the processes in this design is given in appendix 2.

6.2 Simulation results

In appendix 3 the output of the simulator is given. Since the execution of the processes are
recursive, the output presented in appendix 3 covers only the time interval from simulation time 0 to 15. This interval gives us a good indication of the non-deterministic behaviour of the selecting mechanism.

At simulation time 2 there two requests in requestsToSchedule. The scheduler selects the communication between A and C. The communication requests between B and C is invalidated and removed from requestsToSchedule. It does not return in this set at simulation time 3. At simulation time 5, again requestsToSchedule contains a communication request between A and C and a communication request between B and C. This time the communication between B and C is selected and the communication between A and C is dropped removed requestsToSchedule. At simulation time 8 the communication between A and C is selected. At simulation time 11 again the communication between A and C is selected and the communication requests between B and C is dropped. At simulation time 14 the communication between B and C is selected.

We see that the results are just what we wanted. The scheduler used as selection mechanism selects the alternative in a non-deterministic way.
Chapter 7

Future research

A begin is made to design and implement a simulator for POOSL. There is still a lot to do before it is finished. Future research and development has to be done in the following area's:

- Simulator input.
- Simulator output.
- Graphical user interface.
- Interactive simulation.

Concerning the simulation input, a parser-code-generator must be developed that can be used to automate conversion of POOSL descriptions into Smalltalk equivalent. The rules for the code generation are offered in chapter 4. The output of the parser can be used as input for the simulator.

Besides input from file, the possibility to use graphical input has to be investigated and implemented. The graphical and interactive input of IDaSS [Ver90] can be used as an example.

Interactive simulation is another subject to be implemented. The user must be able to make choices during the simulation of a design. In this way, the user is involved by the simulation, and can determine the simulation progress. For example, if there are requests to choose from, the user can influence the choice. The alternatives from which can be chosen, can be visualized as blinking objects on the screen. A choice can be made by clicking on one of these blinking objects. To realize this, a graphical user interface to draw a model of a is necessary.

Simulator output is also a subject for research. We saw that for interactive simulation a graphical user interface is needed. This graphical interface can also be used to visualize the simulation progress. For instance, scheduled communication between processes can be visualized, just like communications in execution, by colouring the channels used for the communication. Once a graphical user interface implemented, a lot more interesting things that can be visualized.

The option to store the simulation results or the design itself in a file must be added to the simulator.

Also the mapping of the cluster classes and the Interrupt- and Abort-statement must be described. These are still examined.
Chapter 8

Conclusions

In this thesis the mapping of POOSL specifications onto Smalltalk equivalents is discussed. Also the conversion of the POOSL syntax into Smalltalk equivalents are given.

POOSL and Smalltalk are very similar to each other, which makes the mapping of POOSL onto Smalltalk not too difficult. One of the difficulties was the mapping of the communication mechanism used in POOSL onto Smalltalk. By adding some additional objects in the Smalltalk equivalent, such as the scheduler and the channels, a useful and working solution is given for this problem.

A class hierarchy is designed, which is used to implement the backbone for the communication between process objects. It can be used to simulate all communications that can be specified in POOSL from simple communications without data transfer to guarded statement with a conditional message-receive statement. Selecting one alternative out of a number of alternatives is also implemented.

Conversion rules for almost all POOSL process statements into Smalltalk equivalent is given. These rules are tested thoroughly. All the conversion rules given in this thesis work properly.

A number of test designs were used to test the design and implementation of our simulator. With these tests the whole trajectory from converting the POOSL syntax into a Smalltalk equivalent up to the actual simulation is tested. According to the simulation results and what we expected, we can conclude that what is implemented until now is working.

This first step towards the design and implementation of the POOSL simulator can be used as a starting-point for designing and implementing a good simulator for POOSL.
References

[Dig88] Digitalk Inc.,

[Dig91] Digitalk Inc.,

Design of a prototype system behaviour simulator for IDaSS.
Master's Thesis.

[PVS95] Putten, P.H.A. van der and J.P.M. Voeten, M.P.J. Stevens
Object-Oriented Co-design for Hardware/Software Systems.

[Sha91] Shafer, D. and D.A. Ritz,
Practical Smalltalk, using Smalltalk/V.

[Ver90] Verschueren, A.C.
IDaSS for ULSI.

[Voe95a] Voeten, J.P.M.
EUT Report 95-E-290

[Voe95b] Voeten, J.P.M.
Appendix 1

The POOSL class hierarchy

Protocol summary for class PScheduler.

Definition
The hierarchy for PScheduler is:
Object
  POOSL
    PScheduler

Private data
Class PScheduler has three instance variables:
requestsToSchedule : an instance of Set to collect incoming requests and requests which has to be scheduled.
scheduledRequests  : an instance of PRequestsList to hold requests which are scheduled.
time                : an instance of Number indicating the current simulation time. It is initialized to have value zero.

Shared data
Class PScheduler has one class variable.
Active : Indicates the active scheduler.

Pool data
Class PScheduler has no pooldictionaries.

Class methods
Class PScheduler has two class methods.
active
   Answer the value of the class variable Active.
inactive
   Inactivate current scheduler by making Active nil.

Instance methods
initialize
   Initializes a new instance of PScheduler to have empty instance variables, scheduledRequests and requestsToSchedule, and initializes the instance variable time to zero.
ExecuteCurTimeReg
   Execute the requests in the set scheduledRequests set.
Step
   Increments the simulation time and handles the requests in the instance variables
Protocol summary for class PChannel.

Class PChannel represent a channel in POOSL a specification.

Definition
The hierarchy for PScheduler is:

Object
POOSL
PChannel

Private data
Class PChannel has five instance variables:

receiveRequests : An instance of Set to collect all incoming receive requests.
sendRequests : An instance of Set to collect all incoming send requests.
name : An instance of String, indicating the channel's name.
transTime : An instance of Number, indicating the time needed to send a message through the channel.
free : An instance of Boolean to indicate whether a channel is in use.

Shared data
Class PChannel has no class variable.

Pool data
Class PChannel has no pooldictionaries.

Class methods
Class PChannel has no class methods
Appendix 1: The POOSL class hierarchy

Instance methods

block
Sets the instance instance variable free to false.
free
Unblocks the channel by setting the instance variable free to true.
initialize
initializes a new instance of PChannel to have empty instance variables sendRequests and receiveRequests, initializes the instance variable transTime to zero, and initialize the instance variable free to true.
isFree
Answers the value of the instance variable free.
name
Answers the value of the instance variable name.
name:aString
Sets the value of the instance variable name to aString.
pair:aPsendRequest with:aPreceiveRequest receive:aPreceiveRequest
Creates an instance of the class PComRequest and adds it to the instance variable requestsToSchedule of the scheduler.
send:aPsendRequest
Accepts and stores the request aPsendrequest, searches for matching receive request.
transTime
Answers the value of the instance variable transTime.
transTime:aTime
Sets the value of the instance variable transTime to aTime.

Protocol summary for class PReceiveRequest.

Class PReceiveRequest represent the request of a process to receive a message.

Definition
The hierarchy for PReceiveRequest is:

Object
POOSL
PRequest
PReceiveRequest

Private data
Class PReceiveRequest has seven instance variables:
name : An instance of String indicating the message name.
guardExpression : A reference to a block of code representing the guard of a guarded command.
Appendix 1: The POOSI class hierarchy

requestor : A reference to an instance of a subclass of PProcess referring to the requesting process.

valid : An instance of Boolean indicating the validity of the request.

nextRequest : A reference to an instance of a subclass of PRequest referring to the next request of the requestor.

channel : A reference to an instance of PChannel referring to the channel used to receive the message.

condition : A reference to a block of code representing the condition for communication.

Shared data
Class PReceiveRequest has no class variable.

Pool data
Class PReceiveRequest has no pool dictionaries.

Class methods
Class PReceiveRequest has no class methods

Instance methods
arguments
   Answers the value of the instance variable arguments.

cannel:aPchannel
   Sets the value of the instance variable channel to aPchannel.

execute
   Executes this receive request.

guardExpression
   Answers the reference to the block, implementing the guard of a guard.

guardExpression:anExpression
   Sets the reference to the block implementing the guard for this statement request.

initialize
   initializes a new instance of PReceiveRequest to have empty instance variables sendRequests and receiveRequests, and initializes the instance variable transTime to zero, and initialize the instance variable valid to true.

invalidate
   invalidates this request and all causally related requests.

isValid
   Answers the value of the instance variable valid.

name
   Answers the value of the instance variable name.

name:aString
   Sets the value of the instance variable name to aString.

nextRequest:aPRequest
   Sets the value of the instance variable nextRequest to aPRequest.
Appendix I: The POOSL class hierarchy

number
  Answers the value of the instance variable number.

dnumber:aNumber
  Sets the value of the instance variable number to aNumber.

requestor
  Answers the value of the instance variable requestor.

requestor:aPProcess
  Sets the value of the instance variable requestor to aPProcess.

validate
  Sets the value of the instance variable valid to true.

Protocol summary for class PSendRequest.

Class PSendRequest represent the send request performed by a process.

Definition
  The hierarchy for PSendRequest is:
    Object
      POOSL
        PRequest
          PSendRequest

Private data
  Class PSendRequest has seven instance variables:

  name : An instance of String indicating the message name.

  guardExpression : A reference to a block of code representing the guard of a guarded command.

  requestor : A reference to an instance of a subclass of PProcess referring to the requesting process.

  valid : An instance of Boolean indicating the validity of the request.

  nextRequest : A reference to an instance of a subclass of PRequest referring to the next request of the requestor.

  channel : A reference to an instance of PChannel referring to the channel used to receive the message.

  condition : A reference to a block of code representing the condition for communication.

Shared data
  Class PSendRequest has no class variable.

Pooldata
  Class PSendRequest has no pooldictionaries.
Class methods
Class PSendRequest has no class methods

Instance methods

arguments
   Answers the value of the instance variable arguments.
arguments:anArray
   Sets the value of the instance variable arguments to anArray.

eexecute
   Executes this send request.
guardExpression
   Answers the reference to the block, implementing the guard of a guard.
guardExpression:anExpression
   Sets the reference to the block implementing the guard for this statement request.
initialize
   Initializes a new instance of PSendRequest to have empty instance variables sendRequests
   and receiveRequests, and initializes the instance variable transTime to zero, and initialize
   the instance variable valid to true.
invalidate
   Invalidates this request and all causally related requests.
isValid
   Answers the value of the instance variable valid.
name
   Answers the value of the instance variable name.
name:aString
   Sets the value of the instance variable name to aString.
nextRequest:aPrequest
   Sets the value of the instance variable nextRequest to aPRequest.
number
   Answers the value of the instance variable number.
number:aNumber
   Sets the value of the instance variable number to aNumber.
requestor
   Answers the value of the instance variable requestor.
requestor:aPprocess
   Sets the value of the instance variable requestor to aPProcess.
validate
   Sets the value of the instance variable valid to true.

Protocol summary for class PComRequest.

Class PComRequest represents a communication request between two processes.
**Appendix 1: The POOSL class hierarchy**

**Definition**
The hierarchy for *PComRequest* is:

Object
POOSL
PRequest
PComRequest

**Private data**
Class *PComRequest* has four instance variables:

- **sendRequest**: A reference to an instance of *PSendRequest*, i.e. the send request belonging to this communication request.
- **receiveRequest**: A reference to an instance of *PReceiveRequest*, i.e. the receive request belonging to this communication request.
- **channel**: A reference to an instance of *PChannel*, i.e. the channel used for communication.
- **valid**: An instance of *Boolean* indicating whether this communication request is still valid or not.

**Shared data**
Class *PComRequest* has no class variable.

**Pooldata**
Class *PComRequest* has no pooldictionaries.

**Class methods**
Class *PComRequest* has no class methods

**Instance methods**

- **channel**
  Answers the value of the instance variable *channel*.
  **channel:aPchannel**
  Sets the value of the instance variable *channel* to aPchannel.

- **execute**
  Executes this request and invalidate this request after execution.

- **initialize**
  Initializes a new instance of *PComRequest* with instance variable *valid* set to true.

- **invalidate**
  Invalidates this request and all causally related requests.

- **isValid**
  Answers the value of the instance variable *valid*.

- **receiveRequest:aPreceiveRequest**
  Sets the value of the instance variable *receiveRequest* to aPRequest.

- **sendRequest:aPSendRequest**
  Sets the value of the instance variable *sendRequest* to aPSendRequest.
Protocol summary for class PWaitRequest.

Class PWaitRequest represent the wait request performed by an process.

Definition
The hierarchy for PWaitRequest is:

Object
  POOSL
  PRequest
    PWaitRequest

Private data
Class PWaitRequest has four instance variables:

requestor : A reference to an instance of PProcess which is the requestor.
valid : An instance of Boolean indicating whether this wait request is still valid.
nextRequest : A reference to an instance of a subclass of PRequest representing the next request of the requestor.
number : An instance of Number indicating the number of this request.

Shared data
Class PWaitRequest has no class variable.

Pooldata
Class PWaitRequest has no pooldictionaries.

Class methods
Class PWaitRequest has no class methods.

Instance methods
initialize
  Initializes a new instance of PWaitRequest with instance variable valid set to true.
invalidate
  Invalidate this request and all causally related requests.
isValid
  Answer the value of the instance variable valid.
nextRequest:aPrequest
  Sets the value of the instance variable nextRequest to aPRequest.
number
  Answers the value of the instance variable number.
number:aNumber
The power doH

Sets the value of the instance variable number to aNumber.

requestor

Answers the value of the instance variable requestor.

requestor:aPprocess

Sets the value of the instance variable requestor to aPProcess.

Protocol summary for class PStatementRequest.

Class PStatementRequest represent the wait request performed by an process.

Definition

The hierarchy for PStatementRequest is:

Object
  POOSL
  PRequest
  PStatementRequest

Private data

Class PStatementRequest has five instance variables:

name : An instance of String holding the name of the message.
guardExpression : A reference to a block of code representing the guard of a guarded command.
requestor : A reference to an instance of a subclass of PProcess referring to the requesting process.
valid : An instance of Boolean indicating the validity of the request.
nextRequest : A reference to an instance of a subclass of PRequest referring to the next request performed by the requestor.

Shared data

Class PStatementRequest has no class variable.

Pooldata

Class PStatementRequest has no pooldictionaries.

Class methods

Class PStatementRequest has no class methods.

Instance methods

execute

Executes this statement requests.

guardExpression

Answers the reference to the block, implementing the guard of a guard.
Appendix 1: The POOSL class hierarchy

guardExpression:anExpression
Sets the reference to the block implementing the guard for this statement request.
initialize
Initialize this statement request.
invalidate
Invalidates this request and all causally related request.
isValid
Is the request still valid? Answer it to the receiver.
name
Sets the name of this request.
name:aString
Answers the name of this request to the receiver.
nextRequest
Answers the reference to the neighbour of this request to the receiver.
nextRequest:aPRequest
Sets the reference to the neighbour of this request.
number
Answers the value of the instance variable number of this request.
number:aNumber
Sets the value of the instance variable number to aNumber.
requestor
Answers the requestor of this request.
requestor:aPProcess
Sets the requestor of this request.
validate
Makes this request valid.

Protocol summary for class PStatementRequest.

Class PScheduledComReq represent a communication request which is actually going to be executed.

Definition
The hierarchy for PScheduledComReq is:
Object
POOSL
PRequest
PScheduledComReq

Private data
Class PStatementRequest has five instance variables:
requestToSend : A reference to an instance of the class PSendRequest referring to the send request.
Appendix 1: The POOSL class hierarchy

nextRequest : A reference to an instance of the class of PReceiveRequest referring to the receive request.

Shared data
Class PScheduledComReq has no class variable.

Pooldata
Class PScheduledComReq has no pool dictionaries.

Class methods
Class PScheduledComReq has no class methods.

Instance methods
execute
   Execute this scheduled communication request. statementrquest.
initialize
   Initialize this communication request.
requestToReceive: aPReceiveRequest
   Sets the instance variable requestToReceive to aPReceiveRequest.
requestToSend: aPSendRequest
   Sets the instance variable requestToSend to aPSendRequest.
Appendix 2

Specification of test design

POOSL specification of process D:

Process class
ProcessD
Instance variables
"
Communication channels
ch3
Message interface
ch3?ProcAOk
ch3?ProcBOk
ch3?TimeOut
Initial method call
startUp
Instance methods
startUp
||
Sel
ch3?ProcAOk
or
ch3?ProcBOk
or
Ch3?TimeOut
les
startUp

Smalltalk equivalent
PPROCESS subclass: #PProcessD
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''!

!PProcessD class methods !!

!PProcessD methods !

startUp
Appendix 2: Specification of test design

"start this process"
| receiveReq1 receiveReq2 receiveReq3 requestExecuted data |

"perform ch1?ok or ch1?timeout"
receiveReq1:= PReceiveRequest new initialize; name:'ProcA Ok'; requestor:self;
   number:1; channel:ch2.
receiveReq2:= PReceiveRequest new initialize; name:'ProcB Ok'; requestor:self;
   number:2; channel:ch2.
receiveReq3:= PReceiveRequest new initialize; name:'Timeout'; requestor:self;
   number:3; channel:ch2.
receiveReq1 nextRequest:receiveReq2.
receiveReq2 nextRequest:receiveReq3.
receiveReq3 nextRequest:receiveReq1.
ch2 receive:receiveReq1.
ch2 receive:receiveReq2.
ch2 receive:receiveReq3.
self wait.
data:= mailBox removeFirst.
requestExecuted:= mailBox removeFirst.
self startUp!!

POOSL specification of process C:

Process class
   ProcessC
Instance variables
   "
Communication channels
   ch1
Message interface
   ch1?ProcAOk
   ch1?ProcBOk
Initial method call
   startUp
Instance methods
   startUp
   | |
   Sel
   ch1?ReceiveA;
   ch3!ProcAOk
   or
   ch1?ReceiveB;
   ch3!ProcBOk
   or
   delay(3);
Smalltalk equivalent

PProcess subclass: #PProcessC
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: "" !

!PProcessC class methods !!

!PProcessC methods!

startUp

"start this process"
| waitReq receiveReq1 receiveReq2 sendRequest requestExecuted data temp delay |
"perform ch1?receiveReq1 or ch3?receiveReq2 or delay of 3 simulation steps"
delay: = 6.
receiveReq1: = PReceiveRequest new initialize;name: 'Receive A';requestor:self;
number:1; channel:ch1.
receiveReq2: = PReceiveRequest new initialize;name: 'Receive B';requestor:self;
number:2; channel:ch1.
waitReq: = PWaitRequest new initialize;requestor:self;number:3.
receiveReq1 nextRequest: receiveReq2.
receiveReq2 nextRequest: waitReq.
waitReq nextRequest: receiveReq1.
ch1 receive: receiveReq1.
ch1 receive: receiveReq2.
PScheduler active scheduleRequest: waitReq after: delay.
self wait.
data: = mailBox removeFirst.
requestExecuted: = mailBox removeFirst.
(requestExecuted = 1)
ifTrue:
   sendRequest: = PSendRequest new initialize;name: 'ProcA Ok';requestor:self;
      number:4; arguments:#(ProcA Ok); channel:ch2.
   sendRequest nextRequest: sendRequest.
   ch2 send: sendRequest.
   self wait.
   requestExecuted: = mailBox removeFirst
] ifFalse:
   (requestExecuted = 2)
Appendix 2: Specification of test design

ifTrue:
  sendRequest: = PSendRequest new initialize; name: 'ProcB Ok'; requestor: self;
  number: 4; arguments: #(ProcB Ok); channel: ch2.
  sendRequest nextRequest: sendRequest.
  ch2 send: sendRequest.
  self wait.
  requestExecuted
 ]
ifFalse:
  sendRequest: = PSendRequest new initialize; name: 'Timeout'; requestor: self;
  number: 4; arguments: #(Timeout); channel: ch2.
  sendRequest nextRequest: sendRequest.
  ch2 send: sendRequest.
  self wait.
  requestExecuted: = mailBox removeFirst
 ]
].
self startUp ! !

POOSL specification of process B:

Process class
  ProcessB
Instance variables
  "
Communication channels
  ch1
Message interface
  ch1?ProcBOK
Initial method call
  startUp
Instance methods
  startUp
    delay(2);
    ch1?ReceiveB
    startUp

Smalltalk equivalent
PProcess subclass: #PProcessB
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

!PProcessB class methods ! !
POOSL specification of process A:

Process class

ProcessA

Instance variables

Communication channels

channel

Message interface

channel?ProcAOk

Initial method call

startUp

Instance methods

startUp

| |

delay(2);

channel?ReceiveA

startUp

Smalltalk equivalent

PPProcess subclass: #PPProcessA

instanceVariableNames: ''

classVariableNames: ''

poolDictionaries: '' !
Appendix 2: Specification of test design

!PProcessA class methods !

!PProcessA methods!

startUp

"start this process"

| waitRequest sendRequest channel delay requestExecuted |

"perform a delay of 2 simulationstep"

delay: = 2.

waitRequest: = PWaitRequest new initialize:requestor:self;number:1.

waitRequest nextRequest:waitRequest.

PScheduler active schedule Wait:waitRequest after:delay.

self wait.

requestExecuted: = mailBox removeFirst.

"perform ch1!requestA"

sendRequest: = PSendRequest new initialize:name:'Receive A';requestor:self;

number:1;arguments:#{(a A)}; channel:ch1.

sendRequest nextRequest:sendRequest.

ch1 send:sendRequest.

self wait.

" self startUp "! !
Appendix 3

Simulation results

***** Begin of a new simulation *****

Process A did a wait request and is waiting for 2 steps.
Process D did a wait request and is waiting for 2 steps.

***** in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel : 'ch1 (trtime:1)'

Request managed by channel 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1

sendRequests is empty
****** out PChannel.receive: ******

***** in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel : 'ch1 (trtime:1)'

Request managed by channel 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1
Requestor: a PProcessC name: 'Receive B' Number: 2

sendRequests is empty
****** out PChannel.receive: ******

process C did two receive requests and a timeout request of 4 steps.

***** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel : 'ch3 (trtime:2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1

sendRequests is empty
****** out PChannel.receive: ******

***** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel : 'ch3 (trtime:2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcB Ok' Number: 2
Requestor: a PProcessD name: 'ProcA Ok' Number: 1

69
Appendix 3: Simulation results

sendRequests is empty

****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel : 'ch3 (trtime:2)'

  Request managed by channel: 'ch3 (trtime:2)'
  The next requests are present in receiveRequests:
  Requestor: a PProcessD name: 'ProcA Ok' Number: 1
  Requestor: a PProcessD name: 'ProcB Ok' Number: 2
  Requestor: a PProcessD name: 'TimeOut' Number: 3

sendRequests is empty

****** out PChannel.receive: ******

Process D did three receive requests and is waiting

start of simulation

Time: 0
Time: 2

****** in PChannel.send: ******
Request performed: process 'A' is ready to send a message on channel : 'ch1 (trtime:1)'

  Request managed by 'ch1 (trtime:1)'
  The next requests are present in receiveRequests:
  Requestor: a PProcessC name: 'Receive A' Number: 1
  Requestor: a PProcessC name: 'Receive B' Number: 2

  The next requests are in sendRequests:
  Requestor: a PProcessA Name: 'Receive A' Number: 1

  The next requests are present in requestsToSchedule:
  Sender: a PProcessA  Receiver: a PProcessC  Channel: 'ch1 (trtime:1)'

Process A is waiting to communicate

****** in PChannel.send: ******
Request performed: process 'B' is ready to send a message on channel : 'ch1 (trtime:1)'

  Request managed by 'ch1 (trtime:1)'
  The next requests are present in receiveRequests:
  Requestor: a PProcessC name: 'Receive A' Number: 1
  Requestor: a PProcessC name: 'Receive B' Number: 2

  The next requests are in sendRequests:
  Requestor: a PProcessA Name: 'Receive A' Number: 1
  Requestor: a PProcessB Name: 'Receive B' Number: 2

  The next requests are present in requestsToSchedule:
  Sender: a PProcessA  Receiver: a PProcessC  Channel: 'ch1 (trtime:1)'
  Sender: a PProcessB  Receiver: a PProcessC  Channel: 'ch1 (trtime:1)'

Process B is waiting to communicate
Appendix 3: Simulation results

Process: A sends message to: C on channel: ch1 (trtime:1)

Channel ch1 (trtime:1) is blocked.

Time: 3
process A did a wait request and is waiting for 2 steps.

Channel ch1 (trtime:1) is unblocked

Request performed: process 'C' is ready to send a message on channel :ch3 (trtime:2)

Request managed by 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor:a PProcessD name:'ProcA Ok' Number:1
Requestor:a PProcessD name:'ProcB Ok' Number:2
Requestor:a PProcessD name:'TimeOut' Number:3

The next requests are in sendRequests:
Requestor:a PProcessC Name:'ProcA Ok' Number:4

The next requests are present in requestsToSchedule:
Sender:a PProcessC Receiver:a PProcessD Channel:'ch3 (trtime:2)'

Process: C sends message to: D on channel: ch3 (trtime:2)

Channel ch3 (trtime:2) is blocked.

Time: 4
Time: 5

Request performed: process 'A' is ready to send a message on channel :ch1 (trtime:1)

ReceiveRequests is empty

The next requests are in sendRequests:
Requestor:a PProcessA Name:'Receive A' Number:1
Requestor:a PProcessB Name:'Receive B' Number:2

Process A is waiting to communicate

out PProcessC startUp

Request performed: process 'C' is ready to receive a message on channel :ch1 (trtime:1)

Request managed by channel 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor:a PProcessC name:'Receive A' Number:1

The next requests are in sendRequests:
Requestor:a PProcessA Name:'Receive A' Number:1
Requestor:a PProcessB Name:'Receive B' Number:2
Appendix 3: Simulation results

The next requests are present in requestsToSchedule:
Sender:a PProcessA  Receiver:a PProcessC  Channel:'chI (trtime:1)'
****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel:'chI (trtime:1)'

Request managed by channel'chI (trtime:1)'
The next requests are present in receiveRequests:
Requestor:a PProcessC name:'Receive A' Number:1
Requestor:a PProcessC name:'Receive B' Number:2

The next requests are in sendRequests:
Requestor:a PProcessA Name:'Receive A' Number:1
Requestor:a PProcessB Name:'Receive B' Number:2

The next requests are present in requestsToSchedule:
Sender:a PProcessB  Receiver:a PProcessC  Channel:'chI (trtime:1)'
Sender:a PProcessA Receiver:a PProcessC  Channel:'chI (trtime:1)'
****** out PChannel.receive: ******

Process C did two receive requests and a timeout request of 4 steps.

Channel ch3 (trtime:2) is unblocked

Process D received: (ProcA Ok) from process C
****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel:'ch3 (trtime:2)'

Request managed by channel'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor:a PProcessD name:'ProcA Ok' Number:1

sendRequests is empty
****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel:'ch3 (trtime:2)'

Request managed by channel'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor:a PProcessD name:'ProcA Ok' Number:1
Requestor:a PProcessD name:'ProcB Ok' Number:2

sendRequests is empty
****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel:'ch3 (trtime:2)'

Request managed by channel'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor:a PProcessD name:'ProcB Ok' Number:2
Requestor:a PProcessD name:'TimeOut' Number:3
Appendix 3: Simulation results

Requestor: a PProcessD name: 'ProcA Ok' Number: 1

sendRequests is empty
****** out PChannel.receive: ******

Process D did three receive requests and is waiting

Process: B sends message to: C on channel: ch1 (trtime: 1)

Channel ch1 (trtime: 1) is blocked.

Time: 6
process D did a wait request and is waiting for 2 steps.

Channel ch1 (trtime: 1) is unblocked

****** in PChannel.send: ******
Request performed: process 'C' is ready to send a message on channel: 'ch3 (trtime: 2)'

Request managed by 'ch3 (trtime: 2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcB Ok' Number: 2
Requestor: a PProcessD name: 'TimeOut' Number: 3
Requestor: a PProcessD name: 'ProcA Ok' Number: 1

The next requests are in sendRequests:
Requestor: a PProcessC Name: 'ProcB Ok' Number: 4

The next requests are present in requestsToSchedule:
Sender: a PProcessC Receiver: a PProcessD Channel: 'ch3 (trtime: 2)'

Process: C sends message to: D on channel: ch3 (trtime: 2)

Channel ch3 (trtime: 2) is blocked.

Time: 8
****** out PProcessC startUp ******

****** in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel: 'ch1 (trtime: 1)'

Request managed by channel 'ch1 (trtime: 1)'
The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1

The next requests are in sendRequests:
Requestor: a PProcessA Name: 'Receive A' Number: 1

The next requests are present in requestsToSchedule:
Sender: a PProcessA Receiver: a PProcessC Channel: 'ch1 (trtime: 1)'

****** out PChannel.receive: ******
****** in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel: 'ch1 (trtime: 1)'

73
Appendix 3: Simulation results

Request managed by channel 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1
Requestor: a PProcessC name: 'Receive B' Number: 2

The next requests are in sendRequests:
Requestor: a PProcessA Name: 'Receive A' Number: 1

****** out PChannel.receive: ******

process C did two receive requests and a timeout request of 4 steps.

Channel ch3 (trtime:2) is unblocked

Process D received: (ProcB Ok) from process C

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel 'ch3 (trtime:2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1

sendRequests is empty

****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel 'ch3 (trtime:2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1
Requestor: a PProcessD name: 'ProcB Ok' Number: 2

sendRequests is empty

****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel 'ch3 (trtime:2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1
Requestor: a PProcessD name: 'ProcB Ok' Number: 2
Requestor: a PProcessD name: 'TimeOut' Number: 3

sendRequests is empty

****** out PChannel.receive: ******

Process D did three receive requests and is waiting

****** in PChannel.send: ******
Request performed: process 'B' is ready to send a message on channel 'ch1 (trtime:1)'

Request managed by 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Appendix 3: Simulation results

Requestor:a PProcessC name:'Receive A' Number:1
Requestor:a PProcessC name:'Receive B' Number:2

The next requests are in sendRequests:
Requestor:a PProcessA Name:'Receive A' Number:1
Requestor:a PProcessB Name:'Receive B' Number:2

The next requests are present in requestsToSchedule:
Sender:a PProcessB Receiver:a PProcessC Channel:'ch1 (trtime:1)'
Sender:a PProcessA Receiver:a PProcessC Channel:'ch1 (trtime:1)'

Process B is waiting to communicate
Process: A sends message to: C on channel: ch1 (trtime:1)
Channel ch1 (trtime:1) is blocked.

Time: 9
process A did a wait request and is waiting for 2 steps.

Channel ch1 (trtime:1) is unblocked

***** in PChannel.send: *****
Request performed : process 'C' is ready to send a message on channel :'ch3 (trtime:2)'
Request managed by 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor:a PProcessD name:'ProcA Ok' Number:1
Requestor:a PProcessD name:'ProcB Ok' Number:2
Requestor:a PProcessD name:'TimeOut' Number:3

The next requests are in sendRequests:
Requestor:a PProcessC Name:'TimeOut' Number:4

The next requests are present in requestsToSchedule:
Sender:a PProcessC Receiver:a PProcessD Channel:'ch3 (trtime:2)'

Process: C sends message to: D on channel: ch3 (trtime:2)
Channel ch3 (trtime:2) is blocked.

Time: 11

***** in PChannel.send: *****
Request performed : process 'A' is ready to send a message on channel :'ch1 (trtime:1)'
Request managed by 'ch1 (trtime:1)'
receiveRequests is empty

The next requests are in sendRequests:
Requestor:a PProcessB Name:'Receive B' Number:2
Requestor:a PProcessA Name:'Receive A' Number:1

Process A is waiting to communicate

***** out PProcessC startUp *****
Appendix 3: Simulation results

******* in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel 'ch1 (trtime:1)'

    Request managed by channel 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1

The next requests are in sendRequests:
Requestor: a PProcessB Name: 'Receive B' Number: 2
Requestor: a PProcessA Name: 'Receive A' Number: 1

The next requests are present in requestsToSchedule:
Sender: a PProcessA Receiver: a PProcessC Channel: 'ch1 (trtime:1)'
******* out PChannel.receive: ******

******* in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel 'ch1 (trtime:1)'

    Request managed by channel 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1
Requestor: a PProcessC name: 'Receive B' Number: 2

The next requests are in sendRequests:
Requestor: a PProcessB Name: 'Receive B' Number: 2
Requestor: a PProcessA Name: 'Receive A' Number: 1

The next requests are present in requestsToSchedule:
Sender: a PProcessB Receiver: a PProcessC Channel: 'ch1 (trtime:1)'
Sender: a PProcessA Receiver: a PProcessC Channel: 'ch1 (trtime:1)'
******* out PChannel.receive: ******

process C did two receive requests and a timeout request of 4 steps.

Channel ch3 (trtime:2) is unblocked

Process D received: (Timeout) from process C
******* in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel 'ch3 (trtime:2)'

    Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1

    sendRequests is empty
******* out PChannel.receive: ******

******* in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel 'ch3 (trtime:2)'

    Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcB Ok' Number: 2

76
Appendix 3: Simulation results

Requestor: a PProcessD name: 'ProcA Ok' Number: 1

sendRequests is empty
****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel: 'ch3 (trtime:2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1
Requestor: a PProcessD name: 'ProcB Ok' Number: 2
Requestor: a PProcessD name: 'TimeOut' Number: 3

sendRequests is empty
****** out PChannel.receive: ******

Process D did three receive requests and is waiting

Process: A sends message to: C on channel: ch1 (trtime: 1)
Channel ch1 (trtime: 1) is blocked.

Time: 12
process A did a wait request and is waiting for 2 steps.
Channel ch1 (trtime: 1) is unblocked

****** in PChannel.send: ******
Request performed: process 'C' is ready to send a message on channel: 'ch3 (trtime:2)'

Request managed by 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1
Requestor: a PProcessD name: 'ProcB Ok' Number: 2
Requestor: a PProcessD name: 'TimeOut' Number: 3
The next requests are in sendRequests:
Requestor: a PProcessC Name: 'TimeOut' Number: 4

The next requests are present in requestsToSchedule:
Sender: a PProcessC Receiver: a PProcessD Channel: 'ch3 (trtime:2)'
Process: C sends message to: D on channel: ch3 (trtime: 2)
Channel ch3 (trtime: 2) is blocked.

Time: 14
****** out PProcessC startUp ******

****** in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel: 'ch1 (trtime:1)'

Request managed by channel 'ch1 (trtime:1)'

77
Appendix 3: Simulation results

The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1

The next requests are in sendRequests:
Requestor: a PProcessB Name: 'Receive B' Number: 2

****** out PChannel.receive: ******

****** in PChannel.receive: ******
Request performed: process 'C' is ready to receive a message on channel : 'ch1 (trtime: 1)'

Request managed by channel 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor: a PProcessC name: 'Receive A' Number: 1
Requestor: a PProcessC name: 'Receive B' Number: 2

The next requests are in sendRequests:
Requestor: a PProcessB Name: 'Receive B' Number: 2

The next requests are present in requestsToSchedule:
Sender: a PProcessB Receiver: a PProcessC Channel: 'ch1 (trtime: 1)'

****** out PChannel.receive: ******

process C did two receive requests and a timeout request of 4 steps.

Channel ch3 (trtime: 2) is unblocked

Process D received: (Timeout) from process C

****** in PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel : 'ch3 (trtime: 2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1

sendRequests is empty

****** out PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel : 'ch3 (trtime: 2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcA Ok' Number: 1
Requestor: a PProcessD name: 'ProcB Ok' Number: 2

sendRequests is empty

****** out PChannel.receive: ******
Request performed: process 'D' is ready to receive a message on channel : 'ch3 (trtime: 2)'

Request managed by channel 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor: a PProcessD name: 'ProcB Ok' Number: 2
Requestor: a PProcessD name: 'TimeOut' Number: 3
Process D did three receive requests and is waiting

Request performed: process 'A' is ready to send a message on channel :ch1 (trtime:1)'

Request managed by 'ch1 (trtime:1)'
The next requests are present in receiveRequests:
Requestor:a PProcessC name:'Receive A' Number:1
Requestor:a PProcessC name:'Receive B' Number:2

The next requests are in sendRequests:
Requestor:a PProcessA Name:'Receive A' Number:1
Requestor:a PProcessB Name:'Receive B' Number:2

The next requests are present in requestsToSchedule :
Sender:a PProcessB Receiver:a PProcessC Channel:'ch1 (trtime:1)'
Sender:a PProcessA Receiver:a PProcessC Channel:'ch1 (trtime:1)'

Process A is waiting to communicate

Process: B sends message to: C on channel: ch1 (trtime:1)
Channel ch1 (trtime:1) is blocked.

Time: 15
process D did a wait request and is waiting for 2 steps.

Channel ch1 (trtime:1) is unblocked

Request performed: process 'C' is ready to send a message on channel :ch3 (trtime:2)'

Request managed by 'ch3 (trtime:2)'
The next requests are present in receiveRequests:
Requestor:a PProcessD name:'ProcB Ok' Number:2
Requestor:a PProcessD name:'TimeOut' Number:3
Requestor:a PProcessD name:'ProcA Ok' Number:1

The next requests are in sendRequests:
Requestor:a PProcessC Name:'TimeOut' Number:4

The next requests are present in requestsToSchedule :
Sender:a PProcessC Receiver:a PProcessD Channel:'ch3 (trtime:2)'

Process: C sends message to: D on channel: ch3 (trtime:2)
Channel ch3 (trtime:2) is blocked.